Ricardo António de Oliveira Torres
Licenciado em Engenharia Informática

Reordenação de Documentos Através de Técnicas de Aprendizagem Automática

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: João Miguel da Costa Magalhães,
Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente: Prof. DoutorJoão Carlos Gomes Moura Pires

Arguente: Prof. Doutor João Carlos Amaro Ferreira

Vogal: Prof. Doutor João Miguel da Costa Magalhães

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2014

Reordenação de Documentos Através de Técnicas de Aprendizagem Automática

Resumo

Sistemas de gestão documental e de recuperação de informação são hoje ferramentas essenciais para aceder aos grandes volumes de informação disponíveis. O exemplo mais popular deste cenário é o motor de pesquisa Google, que se estimava possuir cerca de 45 milhares de milhões de páginas *Web*, em Março de 2013 [14].

Uma vez que a maioria das pessoas, apenas consultam os primeiros dez resultados duma pesquisa, torna-se crucial conseguir uma boa ordenação das respostas, de forma a permitir que o utilizador veja os resultados contendo informação diversificada, de acordo com as suas preferências e indo ao encontro daquilo que escreveu na pesquisa. Além do objetivo de ordenação segundo a *query* escrita pelo utilizador, também foi tido como objetivo a remoção de documentos similares do topo dos resultados das pesquisas.

Nesta tese, pretendemos investigar o uso de algoritmos de aprendizagem de ordenação de resultados, por forma a aumentar a qualidade dos resultados de topo das pesquisas e analisar algumas maneiras para aumentar a diversidade de informação no topo dos resultados das pesquisas.

Uma aplicação foi desenvolvida no contexto desta tese e foi aplicada a um sistema de pesquisa que foi desenvolvido em contexto empresarial com a Quidgest S.A, sendo que posteriormente irá ser integrada numa plataforma de desenvolvimento rápido de aplicações.

Abstract

Document management and information retrieval systems are now essential tools for accessing large volumes of data available. The most popular example of this scenario is the Google search engine, which has been estimated to index about 45 billion Web pages by March 2013 [14].

Since most people check only the first ten results of a search, it is crucial to achieve a good ordering of the responses allowing the user to see the results containing diversified information in accordance with his preferences and meeting the search terms. Both, sorting the search results according to the written query and the removal of similar documents on the top results of a query were considered as objectives in this thesis.

In order to increase the quality of the top results of the queries and in order to analyse some ways to increase the diversity of information at the top results, we have also investigated the use of learning algorithms for sorting results.

An application was developed in the context of this thesis and was applied to a research system that was developed in a business context with Quidgest SA, and will later be integrated into a rapid application development platform.

Agradecimentos

Em primeiro lugar, queria agradecer ao meu orientador Professor João Magalhães, por me ter dado a oportunidade de participar neste projeto e por me ter introduzido na área da investigação.

Obrigado a todos os colaboradores da empresa Quidgest, em especial ao Rodrigo Serafim e ao Eduardo Marques por todo o apoio dado durante o desenvolvimento da tese e pelo apoio dado na integração nos quadros da empresa.

Queria agradecer aos meus pais e à minha irmã, que me apoiaram ao longo deste período de tempo e que sempre me encorajaram para conseguir ultrapassar todas as dificuldades.

Um grande obrigado ao Carlos Loureiro por ter sido um verdadeiro companheiro durante toda esta jornada e por ter tido sempre paciência e disponibilidade para me ajudar com todas as minhas dúvidas.

Por fim, queria agradecer a todos os meus colegas e professores que me acompanharam ao longo do tempo na Universidade Nova de Lisboa.

Índice

Lista de Figuras

Lista de Tabelas

Glossário

Features	Características de aprendizagem
HITS	Hypertext-Induced Topic Selection
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDF	Inverse Document Frequency
IR	Information Retrieval
JSON	JavaScript Object Notation
MAP	Mean Average Precision
MeSH	Medical Subject Headings
nDCG	Normalized Discount Cumulative Gain
PDF	Portable Document Format
Queries	Pesquisas
REST	Representational State Transfer
RTF	Rich Text Format
TF	Term Frequency
XML	Extensible Markup Language

1

Introdução

Inicialmente, a gestão de informação resumia-se à gestão de documentos físicos, como por exemplo, relatórios, artigos, livros, etc. Com a evolução da tecnologia e com a crescente produção de informação digital, o uso e desenvolvimento de sistemas informáticos de gestão de documentos cresceu exponencialmente.

Um sistema de gestão de documentos é um sistema computacional, que facilita a gestão de grandes coleções de documentos, disponibilizando meios para o armazenamento de informação digital, de uma forma organizada e estruturada. Este tipo de sistemas disponibilizam funcionalidades para o controlo de versões de documentos, gestão de metadados, mecanismos de segurança, indexação da informação relativa a cada documento, etc.

A indexação da informação associada a cada documento proporciona uma melhor performance ao sistema de gestão de documentos, pois estes sistemas tiram grande partido da pesquisa por texto e da extração de metadados dos documentos, pois juntando estas duas funcionalidades, a pesquisa e os respetivos resultados tornam-se mais eficazes e eficientes. Mesmo os documentos físicos começam já a ser digitalizados e introduzidos em sistemas de gestão de documentos, visto existirem grandes vantagens na utilização deste tipo de sistemas para a gestão de grandes coleções de documentos.

Um documento pode ser qualquer tipo de ficheiro digital, contendo informação textual e respetivos metadados. O estado da arte no que aos sistemas de gestão de documentos diz respeito, passa por reunir uma coleção de documentos e de seguida, extrair e indexar a informação relativa à informação textual de cada documento e aos metadados, para permitir uma melhor eficiência em termos da pesquisa.

1.1 Contexto e motivação

Com o rápido desenvolvimento e crescimento da *Web*, a cada dia que passa existe mais informação disponível, sendo que em Março de 2013, existiam cerca de 50 milhares de milhões de páginas na *Web* [14]. Com estes números, conseguimos compreender um pouco melhor um dos maiores problemas dos sistemas de informação: a ordenação dos resultados das pesquisas, devido ao possível grande número de documentos retornados.

Com um grande número de documentos, um resultado com uma má ordenação não é, de forma alguma, útil para o utilizador, pois desta maneira, terá de percorrer uma enorme quantidade de dados para conseguir encontrar aquilo que procura. Os sistemas de pesquisa virados para a *Web*, são os melhores exemplos para esta situação, pois cada *query* pode retornar milhões de páginas. Para combater este problema, são necessários bons métodos de ordenação dos resultados de uma *query*.

Existem documentos que abordam o mesmo tema e consequentemente, todos estes documentos estão relacionados entre si, pelo que quando é realizado uma ordenação dos mesmos, segundo a informação que cada um contém, as suas posições na lista ordenada de documentos, serão bastante semelhantes, pelo que se forem estes, os mais importantes/relacionados com a *query* em questão, irão todos estar no topo dessa lista.

O utilizador ao visualizar os resultados da pesquisa, irá ter documentos que abordam a mesma informação ou que, por exemplo, são similares ou até mesmo duplicados. Como não irá percorrer toda a lista de documentos retornados, mesmo sabendo que estão ordenados, irá apenas visualizar os primeiros 15/20 documentos, imaginando que por exemplo, existem 7 documentos similares/duplicados, praticamente metade, que por serem similares/duplicados, abordam a mesma informação e terão um *score* muito semelhante, ora desta forma o utilizador não terá no topo dos resultados uma informação muito diversificada, pelo que não terá um grande leque de opções de informação.

Não é esperado que um utilizador percorra toda a lista de resultados, por exemplo, uma pesquisa num sistema de gestão documental de uma empresa, pode retornar centenas ou até milhares de documentos, e não é esperado que o utilizador examine documento a documento, para descobrir quais os documentos que mais lhe interessa. Ora este é um grande desafio para os sistemas de informação, pois é necessário que os resultados apresentados correspondam efetivamente às necessidades de informação do utilizador, isto é, é necessário que os mesmos sejam ordenados de acordo com uma *query* textual, o *corpus* de documentos, as características e contexto no qual o utilizador

se encontra inserido. Em abordagens tradicionais, os sistemas de pesquisa retornam os dados diretamente indexados pelos termos de pesquisa, ordenando os mesmos segundo um algoritmo independente de contexto. Esta é uma das limitações dos sistemas de informação, visto que desta forma a lista dos resultados, não tem em conta, por exemplo, as preferências do utilizador, as últimas pesquisas realizadas pelo utilizador, etc, e assim desta forma, os resultados, muito provavelmente, não estarão de acordo com as expectativas do utilizador. No fundo, os sistemas de informação tradicionais assumem um modelo de relevância *booleana*, o que não reflete a expressividade de uma *query* textual próxima da linguagem humana.

1.2 Objetivo

Esta tese contribui para a criação de um sistema de pesquisa, que permite realizar pesquisas aos documentos indexados e onde as respostas às pesquisas dos utilizadores, são retornadas ordenadas segundo um algoritmo de aprendizagem e por um algoritmo que reordena os documentos segundo a diversidade de informação apresentada pelos mesmos.

Mais concretamente, esta tese tem como principal objetivo **realizar a reordenação de documentos retornados por uma pesquisa, através de uma análise mais cuidada da diversidade dos documentos.**

1.3 Arquitetura proposta e contribuições

A figura 1 ilustra uma visão sobre as principais componentes da aplicação, sendo também mostrado o fluxo de dados, entre essas mesmas componentes.

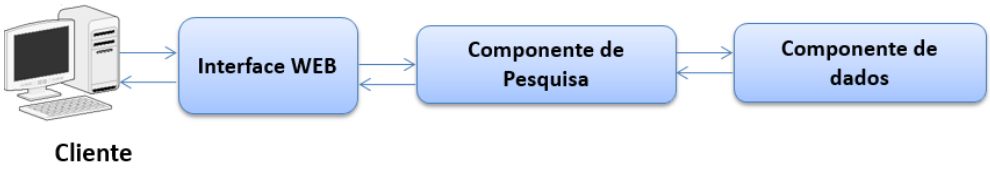

Figura 1. Principais componentes do sistema.

A interface *Web* funciona como porta de entrada dos utilizadores na aplicação, pois as restantes componentes funcionam como caixas neg-ras para os utilizadores. A componente de Pesquisa funciona como uma ponte de ligação entre a interface *Web* e a componente de dados, sendo que possibilita a realização de pesquisas aos

documentos indexados. A componente de dados é a principal componente do sistema, sendo constituída pelas componentes de *Ranking* e de *Re-ranking*, pelo índice que mantém as estruturas necessárias para gerir, indexar e retornar os documentos e os seus metadados associados e pela componente *MySearchHandler*.

Na figura 2 estão ilustrados os elementos da componente de dados e o fluxo de dados entre os mesmos. A componente *MySearchHandler* é responsável pelas pesquisas feitas ao Solr, sendo que esta componente é responsável que a reordenação dos resultados das pesquisas seja feita pela componente de *Re-ranking*.

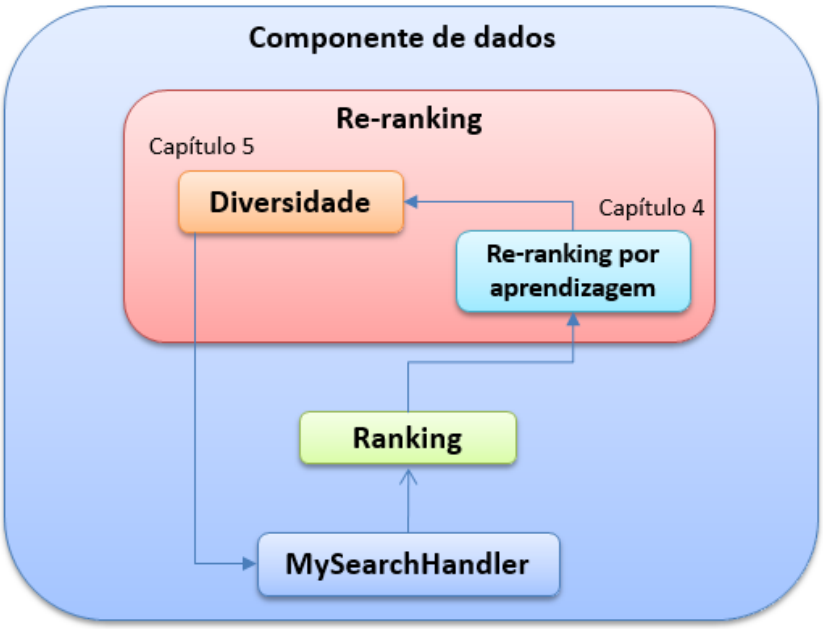

Figura 2. Componente de dados.

A componente *MySearchHandler* envia para a componente de *Ranking* os resultados da pesquisa, sendo responsável por fazer uma primeira ordenação dos resultados da pesquisa. A componente de *Ranking* devolve um subconjunto de documentos, com os cem melhores documentos, de maneira a que a componente de *Re-ranking* consiga reordená-los. Esta componente já estava implementa no próprio Solr, sendo que a componente utiliza algoritmos de reordenação tradicionais, mais concretamente o BM25 [20, 21]. O resultado desta componente, a ordenação simples de todos os documentos do repositório, é passado para a componente de *Re-ranking*. A componente de *Re-ranking* é responsável por realizar a reordenação dos resultados de topo de uma pesquisa, neste caso os cem melhores. Primeiro são reordenados através de um algoritmo de aprendizagem e depois são reordenados através do algoritmo de

diversidade, um dos maiores propósitos desta tese. Por fim, os documentos são enviados novamente para a componente *MySearchHandler*, que envia os resultados para a interface *Web*, para serem visualizados pelo utilizador.

Na figura 3 está ilustrado o fluxo de dados entre as componentes de *Ranking* e *Re-ranking*, na medida em que a primeira, recebe como *input* um conjunto de documentos e *queries* e, o seu output, um subconjunto de documentos ordenados, pertencentes ao conjunto de *input*, é passado à componente de *Re-ranking*, para que a mesma os reordene.

Figura 3. Fluxo de dados entre as componentes de *Ranking* e *Re-ranking*.

As principais contribuições desta tese são:

- **Reordenação por aprendizagem** – utilização de algoritmos de reordenação por aprendizagem para reordenar os cem melhores documentos dos resultados das pesquisas.

- **Diversidade** – desenvolvimento e avaliação de um algoritmo para reordenar o topo dos resultados das pesquisas consoante a diversidade de informação dos documentos, implementado no sistema QSearch, utilizando algoritmos de *locality sensitive hashing* e avaliado utilizando algumas medidas de avaliação de diversidade.

- **Integração com a ferramenta de geração automática de código Genio da Quidgest** – implementação de um *RequestHandler* para o Solr, de maneira a

que a reordenação dos documentos seja feita utilizando os algoritmos de aprendizagem e o algoritmo de diversidade desenvolvido.

1.4 Organização

Esta dissertação está organizada nos seguintes capítulos:

- **Capítulo 2 – Fundamentos:** São apresentados alguns conceitos importantes, como ordenação por métodos clássicos, ordenação por aprendizagem e algumas métricas de avaliação de algoritmos de ordenação, nomeadamente: *Precision*, *MAP* e *nDCG*. São ainda introduzidas algumas tecnologias utilizadas no desenvolvimento desta tese: o Apache Solr e o Apache Tika.

- **Capítulo 3 – Similaridade de documentos:** Este capítulo apresenta a forma como foi utilizado este conceito no contexto desta tese e apresenta os resultados e discussão para a definição do limiar de bits em que dois hashs poderiam diferir para serem considerados ou não similares. É feita também a comparação entre a utilização de dois algoritmos diferentes de *locality sensitive hashing* e também a utilização de *hashs* de trinta e dois bits e sessenta e quatro bits.

- **Capítulo 4 – Reordenação por aprendizagem:** São apresentados alguns conceitos base no contexto dos algoritmos de reordenação por aprendizagem, como por exemplo, como são representados os documentos e alguns algoritmos de reordenação por aprendizagem. São ainda apresentados os testes feitos para a avaliar a performance de alguns algoritmos de reordenação por aprendizagem, duas coleções: OHSUMED e Quidgest.

- **Capítulo 5 – Diversidade:** Este capítulo apresenta a importância que a diversidade de informação tem nos sistemas de recuperação informação atuais. São também apresentadas algumas medidas para medir a diversidade de informação no topo dos resultados das pesquisas, bem como, o algoritmo desenvolvido para aumentar a diversidade de informação no topo dos resultados das pesquisas. São apresentados os testes para medir a performance do algoritmo desenvolvido, utilizando documentos do repositório da Quidgest.

- **Capítulo 6 – Conclusões:** São apresentadas as conclusões desta tese.

2

Fundamentos

2.1 Ordenação por métodos clássicos

Os sistemas de pesquisa têm como requisito retornar a lista dos resultados, ordenada segundo a relevância de cada documento para a pesquisa em questão. Um dos maiores obstáculos, que estes enfrentam é a enorme quantidade de dados que têm de processar, o que faz com que simples abordagens de filtragem de dados não sejam aplicáveis nestes casos. Contudo, estes sistemas utilizam um mecanismo de indexação, para lidar com grandes quantidades de dados, sendo que a técnica mais usada é a do índice invertido.

2.1.1 Indexação

Para realizar as pesquisas, os sistemas de pesquisas mantêm um índice. O processo de indexação contempla quatro fases principais:

1. Recolha de documentos para indexar
2. Conversão do texto para *tokens* indexáveis
3. Pré-processamento linguístico
4. Indexação dos documentos para cada termo/palavra

Na etapa de recolha dos documentos, os mesmos são guardados para serem indexados. Os documentos podem ser recolhidos de três formas: (1) podem estar armazenados num repositório, (2) podem estar referenciados numa base de dados, ou (3) no caso de sistemas de pesquisas *Web*, usando um *parser*.

Na etapa seguinte, a conversão de texto para *tokens*, o conteúdo de cada documento é transformado num conjunto de palavras, através de um processo de normalização e geração de *tokens*. Este processo coloca o conteúdo dos documentos em vetores de texto através dos seguintes passos: (1) conversão de *charset*, (2) remoção de espaços brancos, (3) filtragem/substituição de caracteres e, por fim, o texto passa por um *tokenizer* que o divide segundo uma série de *tokens*.

O pré-processamento linguístico captura alguns padrões de texto e usa algumas heurísticas para minorar o conjunto de palavras de cada documento. Sendo uma das heurísticas usada a remoção de *stopwords*, isto é, palavras que não acrescentam significado ao texto, como por exemplo, *a*, *o*, *de*, *da*. Os sistemas de pesquisa ganham bastante com a remoção destas palavras, pois esta ação resulta numa redução de palavras distintas e consequentemente no tamanho do índice.

Chegando à fase de indexação dos documentos para cada termo/palavra, existe um histograma de representação das palavras para cada documento, que é o resultado do processamento da fase anterior. Com estes histogramas, é construído o índice, sendo que este índice tem de suportar avaliação rápida das *queries*, para poder retornar os documentos que contiverem as palavras presentes na mesma. Para permitir este requisito, os sistemas de pesquisa implementam um índice invertido, que contém um dicionário de todas as palavras e, que permite aceder diretamente à lista de documentos que contém cada palavra.

2.1.2 Modelo do espaço vetorial

Os sistemas de pesquisa não tiram partido da estrutura dos documentos, tratando os mesmos, sempre como *streams* de texto. A maioria dos documentos digitais já são totalmente estruturados, divididos por secções/zonas e contendo alguns metadados. Por exemplo, o nome do autor é um campo que geralmente está presente nos metadados do documento. Para o caso de uma secção do documento, contendo apenas texto, uma solução para guardar a informação é armazenar cada zona do documento no índice invertido, para permitir o acesso direto, podendo optar-se por indexar cada seção codificada, numa espécie de lista associada ao *id* do documento. Desta forma, o tamanho do índice é reduzido e podendo ser feita uma ponderação de zonas. Cada zona tem um peso, sendo que a soma de todos os pesos terá de ser igual a um. Neste sentido, o *score* de um documento *d*, relativamente a uma *query q* é igual à soma dos pesos das zonas que fazem *match* com a *query q*.

O próximo passo, passa por utilizar a frequência de termos para realizar a ordenação

de cada zona, tendo em atenção o número de vezes que cada palavra aparece em cada zona e a relevância que tem para a *query* em questão. Uma das abordagens utilizadas, é pura e simplesmente, a frequência do termo *t*, no documento *d*, denotado por $tf_{t,d}$, sendo definida pela seguinte fórmula:

$$tf_{t,d} = frequency(t,d) \quad (1)$$

Contudo, existem problemas associados a esta abordagem, que passam essencialmente, pela existência de palavras muito comuns e que não são úteis para a conclusão da relevância do documento. Para combater este problema, existe um outro fator, idf_t. Sendo que a seguinte fórmula, espelha como esta abordagem calcula o *score* dos documentos:

$$idf_t = \log\left(\frac{N}{df_t}\right) \quad (2)$$

Sendo que *N*, representa o número total de documentos existentes na coleção e df_t, representa o número de documentos que contêm o termo *t*. Esta abordagem *TF_IDF* combina a frequência de termos, *TF*, e a *inverse document frequency*, *IDF*. O peso do termo *t*, no documento *d* é dado pela seguinte fórmula:

$$tf\_idf_{t,d} = tf_{t,d} \,.\, idf_t \quad (3)$$

Este esquema de ponderação de termos, atribui o maior peso a um termo, se este ocorrer muitas vezes num pequeno número de documentos e atribui o menor peso, se este ocorrer na maior parte dos documentos.

Queries são declarações formais de necessidades de informação, mais concretamente palavras-chave que são pesquisadas nos sistemas de informação. Uma *query* não tem de ter correspondência de apenas um documento da coleção, na verdade vários documentos podem ter correspondência com a pesquisa feita, possuindo esses documentos diferentes graus de relevância.

O *score* de um documento é calculado usando a medida de *score overlap*, modificada para utilizar o *TF_IDF*. Nesta medida, o *score* de um documento *d* é calculado com base na soma do número de ocorrências de cada termo, presente na *query q*, no documento *d*. Na versão modificada para utilizar o *TF_IDF*, o score é calculado com base na seguinte fórmula:

$$\sum_{t \in Q} tf\_idf_{t,d} \tag{4}$$

Este modelo representa documentos através de vetores, no mesmo espaço vetorial. Nos sistemas de pesquisa, isto é feito, assumindo que cada documento é um vetor, contendo uma componente para cada termo, em conjunto com um peso calculado para cada componente de acordo com o modelo de ordenação.

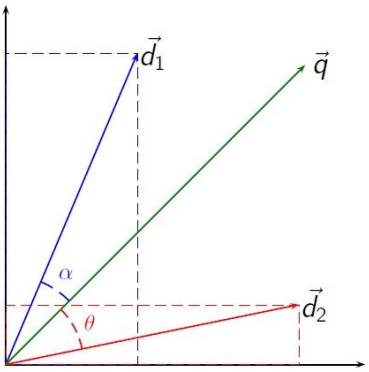

Figura 4. Modelo do espaço vetorial.

Neste modelo, o grau de relevância pode ser calculado com uma precisão ainda maior, pois quando a consulta e o documento são vistos como vetores no mesmo espaço Euclidiano, o produto interno entre a pesquisa e cada documento pode ser utilizado para calcular o grau de semelhança.

Para calcular o *score* de um documento relativo a uma *query*, é usada a distância do cosseno [19]. Sendo que dada uma *query* q e um documento d, a distância do cosseno é dada pela seguinte fórmula:

$$\cos(q, d) = \frac{q \ . \ d}{\|q\| \ . \ \|d\|} \tag{5}$$

2.1.3 Okapi BM25

O algoritmo Okapi BM25 [20, 21], é um dos melhores algoritmos de ordenação de documentos, representando o estado da arte, no que aos algoritmos baseados na frequência de termos para obtenção de documentos diz respeito, há mais de duas décadas. O *score* de cada documento d, dado uma *query* q, é calculado com base na seguinte fórmula:

$$score(d,q) = \sum_{t \epsilon q} \frac{(k+1)\,.\,tf(t,d)}{tf(t,d) + k\,.\left(1 - b + b\,.\frac{|d|}{avgdl}\right)}\,.\,IDF(t) \quad (6)$$

onde $tf(t, d)$ representa a frequência da palavra t no documento d, $|d|$ o número de palavras presentes no documento d e $avgdl$ a média do número de palavras de todos os documentos indexados. k e b são variáveis que são definidas empiricamente.

A função $IDF(t)$ calcula o peso do número de documentos com o termo t, usando a seguinte fórmula:

$$IDF(t) = \log\left(\frac{N - n(t) + 0.5}{n(t) + 0.5}\right) \quad (7)$$

onde N representa o número total de documentos e $n(t)$ o número de documentos que contêm o termo t. Comparando com a fórmula 2, já apresentada anteriormente, neste caso, excluindo a soma de 0.5 no numerador e no denominador, a diferença reside no numerador das duas fórmulas, já que no caso da equação 2 temos o número de documentos da coleção e no caso da equação 7 temos o número de documentos que não contêm o termo *t*, ou seja, pretende-se medir o quão raro é o termo *t* na coleção.

2.2 Ordenação baseada no grafo Web

Existem dois tipos de algoritmos de ordenação de páginas *Web*: (1) baseados nos conteúdos das mesmas e (2) baseados nas referências para outras páginas *Web*. Para esta tese, são mais relevantes algoritmos de ordenação baseados nas referências para cada página, por isso de seguida serão descritos dois dos mais populares algoritmos desta área. Este género de algoritmos tiram partido das conexões entre páginas, isto é, *links* entre páginas *Web*, para aferir qual a importância de cada nó do grafo, sendo que cada página representa um nó desse mesmo grafo.

2.2.1 Algoritmo PageRank

O *score* de uma página *Web* representa a importância da mesma na Internet. O *PageRank* [18] é um algoritmo, desenvolvido pela Google, que tem como objetivo medir a importância das páginas na *Web*. Este algoritmo utiliza o grafo da *Web* para inferir a importância de cada página.

O algoritmo atribui pesos a todos os nós do grafo *Web*, tendo para isso em conta o número de páginas que referenciam a página em questão, o número total de

referências de cada página que referencia a página para a qual está a ser calculado o peso e, por último, o número de páginas que não têm referências para outras páginas. Sendo que a seguinte fórmula tem em conta os três fatores referidos anteriormente:

$$ranking(w_i) = \delta \cdot \left(\sum_{\forall ref(w_j, w_i)} \frac{ranking(w_j)}{num\_ref(w_j)} + \sum_{\gamma \in \tau} \frac{ranking(\gamma)}{n} \right) + (1 - \delta) \qquad (8)$$

onde $\forall ref(w_j, w_i)$ representa todas as páginas w_j que têm uma referência para a página w_i, $num\_ref(w_j)$ indica o número de referências feitas pela página w_j e, por fim, $\gamma \in \tau$, representam as páginas que não têm qualquer referência para nenhuma outra página. δ representa a probabilidade de um utilizador seguir uma referência para uma outra página *Web*. Podendo um utilizador chegar a uma página de uma forma aleatória, a fórmula acima referida tem esse facto em conta, sendo que $(1 - \delta)$ representa esse mesmo fator.

2.2.2 Algoritmo HITS

O HITS [28] é um algoritmo que analisa os *links* entre páginas *Web* para as classificar. A ideia do algoritmo baseia-se na divisão das páginas em dois conjuntos: *Hubs* e *Authorities*. *Authorities* são páginas que contêm informação importante, ao passo que os *Hubs* são páginas que contêm referências para *Authorities*, sendo que a mesma página pode ser ao mesmo tempo, uma *Authority* e um *Hub*. Desta forma, cada nó i tem associado dois *scores*: o *Authority score* (a_i) e o *Hub score* (h_i).

$$a_i = \sum_{j \to i} h_j \qquad\qquad h_i = \sum_{j \to i} a_i \qquad (9)$$

Ambos são inicializados com o valor 1, mas com o avançar das iterações do algoritmo, vão sendo actualizados até chegarem a um ponto em que estabilizam para um determinado valor.

2.3 Ordenação por aprendizagem

Ordenação por aprendizagem é um problema de *machine learning*, cujo objetivo é aprender um modelo de reordenação de documentos de acordo com uma dada *query*. Os métodos de ordenação por aprendizagem permitem que os sistemas de pesquisa

utilizem uma série de características para realizarem a ordenação dos resultados das pesquisas. Com este tipo de ordenação pretendemos combinar as várias características extraídas dos pares (*query*-documento), para obter melhores resultados no que à ordenação dos documentos diz respeito.

2.3.1 Formalização

Os algoritmos de ordenação por aprendizagem são usados para encontrar um modelo de ordenação, usando para isso, dados de treino. Esses dados de treino, contêm *queries* e os documentos associados a cada uma dessas *queries*. O modelo de ordenação é determinado por um algoritmo de ordenação por aprendizagem e o objetivo é que esse modelo seja capaz de fazer uma boa ordenação para novas *queries*.

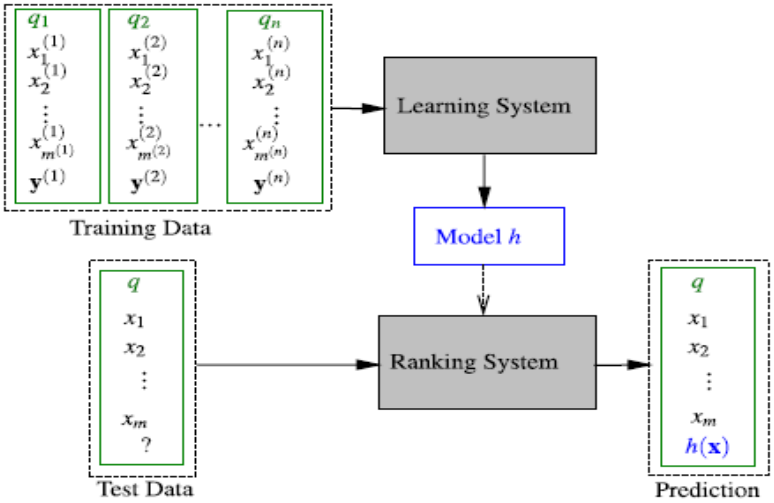

Figura 5. Abordagem utilizada nos métodos de ordenação por aprendizagem. [15]

Na figura 5, $q_i \epsilon \{i = 1, \dots, n\}$, corresponde às n *queries*, usadas para realizar o passo de treino. $\{x_j^i\}_{j=1}^{m(i)}$ representa os vetores de características, associados a cada *query*, enquanto que $y^i \epsilon \{i = 1, \dots, n\}$, corresponde ao conjunto de *scores*.

Quando é iniciado um método de aprendizagem específico para um conjunto de dados de treino, o sistema combina os diferentes estimadores de relevância de uma forma otimizada, de maneira a aprender o modelo de ordenação. Durante o processo de aprendizagem, uma função de custo é aplicada para estimar as inconsistências entre a hipótese h e os vetores de características y. No passo de teste, o modelo de ordenação é aplicado a uma nova *query*, com o intuito de realizar a ordenação dos documentos que fizerem *match* com a mesma.

2.3.2 Tipos de abordagens

Existem três tipos de abordagens de aprendizagem: *Pointwise* [15], *Pairwise* [15] e *Listwise* [16]. No caso *Pointwise*, os graus de relevância dos documentos são guardados como numéricos ou como *scores* ordinais. O problema de ordenação por aprendizagem é visto como um problema de regressão ou classificação. A ideia que esta abordagem tem por base é que dados os vetores de características de cada documento, relativamente aos dados de treino, a relevância de cada documento pode ser medida com base em funções de *score* e, a partir destes *scores*, a lista de documentos pode ser ordenada. Uma função de custo é aplicada, com o intuito de medir eventuais inconsistências entre a função de *score* e os dados de teste.

A vantagem de utilizar a estratégia *Pointwise* passa pelo facto de suportar diretamente a aplicação de teorias e algoritmos existentes na classificação, contudo, a informação relativa à ordem dos documentos não pode ser utilizada no passo de treino, pois os algoritmos recebem um documento de cada vez como *input*.

No caso *Pairwise*, os valores de relevância são guardados como valores binários, que dizem qual a melhor ordem em relação a cada par de documentos. Desta forma, os métodos de ordenação por aprendizagem são vistos como problemas de classificação. A ideia geral é que dados os vetores de características de cada par de documentos, relativamente aos dados de treino, o grau de relevância de cada um dos documentos pode ser calculado com as funções de *score*, que tentam minorar o número de documentos mal classificados. A função de custo usada nesta abordagem, tem em conta a ordem relativa de cada par de documentos.

Existem vários métodos que usam esta estratégia, sendo os mais representativos: *RankNet* [5], *RankBoost* [6] e *SVMrank* [22]. Comparando a abordagem *Pairwise* com a *Pointwise*, o método de aprendizagem opera sobre cada par de documentos, ao passo que na abordagem *Pointwise*, os métodos de aprendizagem, operam sobre cada documento.

Neste caso, tanto a *Pairwise* e a *Pointwise* possuem a mesma vantagem da anterior, relativamente a suportar diretamente a aplicação de teorias e algoritmos existentes na classificação. No entanto, uma das maiores desvantagens, reside no facto de a função de custo apenas ter em conta o grau de relevância de cada par de documentos e não de cada documento.

No caso da *Listwise*, a ordenação por aprendizagem tem em consideração um

conjunto de documentos associados a uma *query* e realiza o treino de uma função de ordenação, através da minimização de uma função de custo *Listwise*, definida nos dois conjuntos de documentos, quer no conjunto de *ouput* previsto, quer no conjunto de dados de teste.

A ideia que esta abordagem tem por base é que dados os vetores de características do conjunto de documentos para treino, o grau de relevância de cada um destes documentos pode ser calculado com o recurso a funções de *score*, que tentam otimizar diretamente o valor de métricas sobre certas informações, fazendo a média de todas as *queries* aos dados de treino.

A função de custo usada neste caso, tem em conta a posição dos documentos na lista, já ordenada, de todos os documentos associados à mesma *query*. Este é um problema de otimização difícil, pois a maioria das medidas de avaliação não são funções contínuas. Nestes casos, têm sido utilizadas aproximações contínuas.

Vários métodos têm sido propostos, mas os mais representativos são: *AdaRank* [7], *ListNet* [10] e *Coordinate Ascent* [16]. A principal vantagem desta abordagem, em comparação com as outras duas abordagens descritas anteriormente, reside no facto de que a função de custo tem em conta as posições dos documentos, numa lista ordenada de todos os documentos associados à consulta. Esta medida, de ser tomada em conta a posição dos documentos, torna a pesquisa mais eficiente.

Este será um tema que será abordado mais aprofundadamente no capítulo 4, onde será descrito mais detalhadamente todo o processo da reordenação por aprendizagem automática.

2.4 Métricas de avaliação

2.4.1 Precision *(P@n)*

Mede a relevância das n posições de topo de uma lista ordenada, representando a fração de documentos que são relevantes nessas primeiras n posições. A sua fórmula de cálculo é a seguinte [25]:

$$P@n = \frac{|\text{documentos relevantes nos n resultados de topo}|}{n} \tag{10}$$

2.4.2 Mean Average Precision (*MAP*)

Esta métrica é definida como sendo a média da *Precision*, sobre o conjunto de todas as

queries e é expressa de acordo com a seguinte fórmula:

$$MAP = \frac{1}{|Q|} \sum_{q=1}^{Q} AP_q \tag{11}$$

sendo que $|Q|$ denota o número de *queries* e AP_q é dado pela seguinte fórmula:

$$AP_q = \frac{\sum_{n=1}^{N_q} P@n \ . \ r_n}{\left|documentos\ relevantes\ query_q\right|} \tag{12}$$

N_q representa o número de documentos retornados e r_n é igual a 1 caso o documento n seja relevante, 0 caso contrário.

2.4.3 Normalized Discount Cumulative Gain (*nDCG@n*)

É uma medida de avaliação das n primeiras posições, de uma lista ordenada, quando existe julgamento de relevância de múltiplos níveis. É definida da seguinte forma [25]:

$$nDCG@n = Z_n \sum_{j=1}^{n} \frac{2^{r_j} - 1}{log(1+j)} \tag{13}$$

sendo que r_j representa o *score* do documento j e Z_n é um fator de normalização, escolhido de tal forma, tal que a ordenação perfeita recebe um *nDCG@n* igual a 1.

2.5 Apache SOLR

O Apache Solr [2] é um servidor de pesquisa, do projeto Apache Lucene. Tem como principais características pesquisa por texto simples ou multifacetada, *clustering* dinâmico de documentos, integração com bases de dados, bem como o processamento e indexação de vários tipos de ficheiros (*doc, pdf,* etc). Outra grande vantagem do Solr é o facto de ser escalável, permitindo, por isso, que sejam efetuadas pesquisas distribuídas e que os índices estejam replicados.

O Solr providencia uma API *REST*, que suporta *xml*, *json* e binário sobre *Http*, para indexação de documentos, bem como para a realização de pesquisas sobre os mesmos. O Solr é uma multiplataforma, sendo que o seu código, escrito na linguagem de programação Java, está disponível como um *software open source*, ao abrigo da licença Apache [1]. Esta ferramenta disponibiliza, ainda, uma interface *Web*, para o administrador do sistema poder gerir o mesmo, de uma forma mais simples e intuitiva.

2.5.1 Apache Lucene: Indexação e Pesquisa

A biblioteca Apache Lucene [4] é o core do Solr e disponibiliza *full-text search*. O Lucene é um sistema de pesquisa e ao mesmo tempo uma API *open source* de indexação de documentos, desenvolvido na linguagem de programação Java.

O Lucene apenas contém o núcleo do motor de pesquisa. Por isso, não tendo incorporado nenhum *Web crawler* ou *parser* para diferentes formatos de documentos, devendo ser o utilizador a adicionar estas funcionalidades, se assim o pretender. Para o Lucene não importa a origem dos dados, o seu formato ou mesmo a linguagem em que está escrito, desde que o conteúdo desses documentos possam ser convertidos para texto. Isto quer dizer que o Lucene pode ser utilizado para indexar e realizar pesquisas sobre os dados gravados em arquivos, páginas *Web*, documentos armazenados no sistema de ficheiros local, documentos de texto, documentos *Microsoft Word*, documentos *html*, documentos *pdf*, ou qualquer outro formato do qual possa ser extraída a sua informação textual.

A biblioteca é composta por duas componentes principais: indexação e pesquisa, tal como pode ser visto na figura 5. A indexação processa os dados originais gerando uma estrutura de dados inter-relacionada, eficiente para a pesquisa, baseada em *keywords*. A pesquisa, por sua vez, consulta o índice pelas palavras escolhidas para a consulta e organiza os resultados pela similaridade do texto com a mesma.

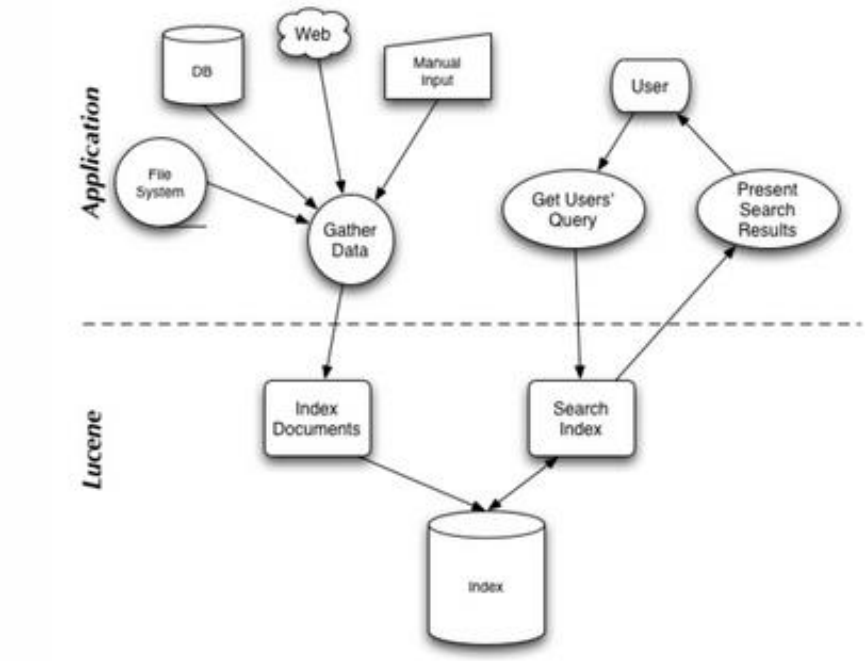

Figura 6. Visão geral da biblioteca Lucene.

O Lucene armazena os dados numa estrutura de dados denominada de índice invertido, que pode ser armazenada em memória ou no sistema de ficheiros local. Oferece um grande nível de abstração, pois o programador não precisa de conhecer funções e algoritmos de indexação, nem de consulta, basta utilizá-los através da API da biblioteca. Apesar da facilidade de uso, o Lucene não implementa um *Web Crawler* ou *parsers* de *html/xml*, como já foi mencionado. Este processamento, se necessário, deve ser feito pela aplicação e passado à biblioteca por meio de instâncias da classe *Document*.

A classe *Document* representa o mecanismo escolhido pelo Lucene para troca de informação entre a aplicação e a biblioteca. Um documento é composto por campos e esses campos por informações. Esses campos são úteis nas pesquisas, de modo a permitir que sejam pesquisados valores específicos para um determinado campo. O resultado da consulta é um conjunto de documentos ordenado por relevância.

O Lucene implementa, ainda, uma linguagem para a realização de *queries*, que proporciona pesquisas *booleanas, "Luis AND Inacio AND Silva OR Ricardo"*, restritivas, *" + George + Rice - eat – pudding"*, por campos, "*animal: monkey AND food: banana"*, por expressões regulares, *"test *, te? t"*, consultas ponderadas, *"jakarta ^ 4 apache"*, e em *range, "date: [20020101 TO 20030101]"*. Além disso, ainda permite ao programador criar

dois tipos de consultas avançadas: (i) *fuzzy*, que utiliza a técnica da distância de *Levenshtein* [29], ao avaliar a proximidade entre a pesquisa e o documento, e permite informar o nível de similaridade mínimo, como, por exemplo, na consulta "*roam ~ 0.8*"; (ii) pesquisa por proximidade entre palavras, como, por exemplo, a consulta *"jakarta apache ~ 10"*, que retornará todos os documentos que possuam as palavras *jakarta* e *apache* com no máximo 10 palavras entre as mesmas.

2.5.2 Fields, Analysers e Tokens

O Solr oferece a possibilidade de escolhermos o modelo de dados, através dos *fields*. Para campos textuais, o *Solr* permite em tempo de indexação, pré processar os dados através de *analysers*. Um *analyser* pode estar associado a um ou mais campos de texto. Sendo que a função de um *analyser* é indicar uma sequência de processamento, a ser aplicado ao texto que se pretende inserir no sistema.

O processamento que é efetuado a esse texto pode ser agrupado em três grandes categorias:

- ***Character Filter*** – Efetua um processamento ao nível das letras do texto. "Consome" e produz uma *stream* com as letras.

- ***Tokenizer*** – Divide o texto segundo *tokens* ou então através de uma expressão regular.

- ***Token Filter*** – Permite manipular os *tokens* processados pelo *tokenizer*, como por exemplo, remover *tokens* duplicados.

De seguida apresentamos um pouco do código da classe *StandardAnalyzer*, por ser uma classe que vai ser importante, nomeadamente, na construção dos *hashs*. O método *tokenStream* é utilizado para obter todos os termos de um documento, mantendo a ordem pela qual os termos aparecem no corpo do texto dos documentos.

```
import org.apache.lucene.analysis.*;
import org.apache.lucene.util.Version;

import java.io.File;
import java.io.IOException;
import java.io.Reader;
import java.util.Set;

public class StandardAnalyzer extends Analyzer {
    private Set<?> stopSet;
    private final boolean replaceInvalidAcronym,enableStopPositionIncrements;

    public static final Set<?> STOP_WORDS_SET = StopAnalyzer.ENGLISH_STOP_WORDS_SET;
    private final Version matchVersion;

    public StandardAnalyzer(Version matchVersion) {
        this(matchVersion, STOP_WORDS_SET);
    }

    public StandardAnalyzer(Version matchVersion, Set<?> stopWords) {
        stopSet = stopWords;
        setOverridesTokenStreamMethod(StandardAnalyzer.class);
        enableStopPositionIncrements = StopFilter.getEnablePositionIncrementsVersionDefault(matchVersion);
        replaceInvalidAcronym = matchVersion.onOrAfter(Version.LUCENE_24);
        this.matchVersion = matchVersion;
    }

    @Override
    public TokenStream tokenStream(String fieldName, Reader reader) {
        StandardTokenizer tokenStream = new StandardTokenizer(matchVersion, reader);
        tokenStream.setMaxTokenLength(maxTokenLength);
        TokenStream result = new StandardFilter(tokenStream);
        result = new LowerCaseFilter(result);
        result = new StopFilter(enableStopPositionIncrements, result, stopSet);
        return result;
    }
```

Figura 7. Parte do código da classe *StandardAnalyzer* do Solr.

2.5.3 Schema

O Solr é uma ferramenta bastante flexível, pois todas as configurações são feitas e mantidas em ficheiros *xml*. As configurações ao nível da indexação e armazenamento são guardadas num ficheiro *xml*, denominado de *Schema*. Este ficheiro, contém informação relativa aos campos existentes, sendo estes campos valores que se pretendem indexar, para depois serem usados nas pesquisas ou na realização de ordenações. Neste ficheiro está, também, informação relativa à chave primária, aos campos que têm de ser únicos, os que são obrigatórios, etc.

Tipos de campos

A unidade fundamental quer do Solr, quer do Lucene são os termos, que são indexados e usados nas pesquisas. Para este efeito, a definição do tipo de campo inclui uma ou mais configurações de *Analyzers*, para controlar os passos de análise de texto, a serem tomados no processamento dos termos do índice ou o tempo de consulta para um determinado tipo de campo. Isto significa que *tokenization* e outras componentes de análise são específicas para cada tipo de campo.

Campos

Os campos são definidos a partir da declaração dos valores dos atributos que compuserem o mesmo. O nome do campo é usado como identificador único e o tipo deve referenciar um tipo de campo que foi declarado anteriormente no *Schema*. O Solr é flexível na definição de campos, sendo que permite, por exemplo, controlar se os dados extraídos são armazenados num campo, com um atributo *booleano*. Existe um grande número de outros atributos *booleanos*, para controlar detalhes de como os dados são armazenados, tais como:

- ***indexed*** – Permite que o campo seja pesquisado e ordenado. Consequentemente, este campo irá ser processado por *analysers* e indexado no índice.
- ***stored*** – Permite o armazenamento dos dados, para que eles possam, mais tarde, serem retornados nos resultados das pesquisas. O campo pode ter o valor false para evitar redundância, no caso dos dados já estarem armazenados noutro(s) campo(s).
- ***multiValued*** – Deve ser *true*, caso o campo possa ter mais do que um valor diferente.

Se o campo possuir a opção de ser indexado, então existem algumas opções adicionais:

- ***omitNorms*** – Esta opção afeta o *score* dos documentos e reduz o uso de memória por campo. Pode ser utilizada, quando o comprimento do campo não deve afetar o *score* e para campos que não serão tomados em conta no *score*.
- ***omitPositions*** – Esta opção desativa consultas por frase, porque omite informações sobre a posição do termo no índice para o campo em questão. Desta forma, permite poupar algum espaço no índice.
- ***omitTermFreqAndPositions*** – Esta opção influencia negativamente a eficiência da pesquisa e não permite *queries* por frase. Omite, ainda, as posições e as frequências dos termos no índice.
- ***termVectors*** – Esta opção garante uma melhor performance em relação à eficiência da pesquisa, para funcionalidades como o *MoreLikeThis*. Esta opção aumentará, muito provavelmente, quer o tamanho do índice, quer o tempo de indexação.

Campos dinâmicos

Campos podem ser criados em tempo real, tendo para isso de fazer um *match* com o

nome dos campos, no *Schema*, que têm a declaração de *dynamicFields*. A definição dos campos dinâmicos têm os mesmos atributos e opções de que os campos tradicionais, à exceção do nome que é um *wildcard*. Eles apenas fazem *match*, caso o nome em questão, não fizer *match* com nenhuma definição de campo tradicional.

Chave única

Declara o campo como identificador único para cada documento e auxilia o Solr a verificar se um novo documento inserido, já existe no índice e consequentemente, deve ser atualizado, ou se é novo.

Campos copiados

Permite o copiar os dados de um campo para outro campo, mesmo tendo comportamentos de indexação diferentes. Outra aplicação desta declaração é a prática comum de copiar vários campos para um campo comum, de modo a melhorar o tempo de pesquisa.

Schema do Solr do QSearch

Na figura seguinte mostramos alguns dos campos que foram definidos para o *Schema* do Solr do QSearch.

```
<fields>
   <field name="id" type="string" required="true" indexed="true"
stored="true" multiValued="false" />
    <field name="filename" type="text_general" indexed="true"
stored="true" multiValued="true"/>
    <field name="title" type="text_pt" indexed="true" stored="true"
multiValued="true"/>
    <field name="author" type="text_general" indexed="true"
stored="true"/>
    <field name="hash" type="text_general" indexed="true"
stored="true" multiValued="false"/>
    <field name="tags" type="text_general" indexed="true"
stored="true" multiValued="true"/>
    <field name="category" type="text_general" indexed="true"
stored="true"/>
    <field name="last_modified" type="date" indexed="true"
stored="true"/>
   <field name="duplicates" type="string" indexed="true" stored="true"
multiValued="true"/>
    <field name="body" type="text_pt" indexed="true" stored="true"
multiValued="true"  termVectors="true" termPositions="true"
termOffsets="true"/>
</fields>
```

Figura 8. Definição de alguns campos do Solr do QSearch.

Cada documento possui um identificador único (*id*), o nome do ficheiro (*filename*), um título (*title*), um autor (*author*), a data da última modificação (*last_modified*), o seu conteúdo (*content*), hash do seu conteúdo (*hash*) e a que categoria pertence (*category*). Sendo que cada documento tem também como atributos, os nomes dos seus ficheiros duplicados (*duplicates)*. Visto ser um campo *multiValued*, pode conter mais do que uma referência para outros ficheiros. Desta forma, quando existem entradas de novos ficheiros no sistema, são automaticamente feitas as devidas ligações para os novos ficheiros. Possuí ainda as *tags* introduzidas pelos utilizadores e informação sobre os termos que compõem o conteúdo do próprio (*body*).

2.5.4 Query parsers e ordenação

O Solr disponibiliza uma linguagem para *queries* bastante rica e completa, linguagem essa que permite realizar vários tipos de pesquisa: por texto simples, por valores presentes nos campos, utilizando *wildcards*('*'), pesquisas *fuzzy* ou então pesquisa num intervalo de valores. Sendo que o Solr ainda permite a criação de uma nova linguagem para a realização das pesquisas, para além da existente por defeito.

Quando é realizada uma pesquisa, por cada documento que faz *match* com a pesquisa em questão, é-lhe associado um *score*. Este *score* indica o grau de similaridade do documento em relação à pesquisa efetuada. Este valor é calculado para cada documento d que contenha o termo t, presente na pesquisa q, de acordo com a seguinte fórmula:

$$\sum_{t \epsilon q} \left(tf(t,d) \,.\, IDF(t)^2 \,.\, boost(t.field,d) \,.\, lengthNorm(t.field,d) \right) \qquad (14)$$

Nesta fórmula, é importante realçar o fator $boost\ (t.field,\ d)$, que atribui um valor de impulso a uma palavra ou documento, obtendo desta forma, um melhor ou pior *score* na pesquisa. Este valor fica calculado nos índices, o que torna impossível a sua utilização, no caso de algum fator ser alterado dinamicamente. No entanto, o Solr, sendo uma ferramenta extensível, permite que sejam utilizados outros algoritmos de ordenação, aquando da realização de uma pesquisa.

2.6 Processamento de documentos com estrutura

Existe um pequeno problema associado à indexação de documentos por parte do Solr, pois se quisermos fazer pesquisas sobre algum atributo do documento, por

exemplo, o seu conteúdo, é necessário, primeiro extrair e indexar essa mesma informação. Ora para concretizar este objetivo, visto que o Solr não realiza tal tarefa, recorrer-se-á à ferramenta Apache Tika.

O Apache Tika é uma biblioteca desenhada com o objetivo de extrair informação de documentos estruturados, tais como *pdf*, ficheiros *Microsoft Office*, *rtf*, etc. Com esta ferramenta, é possível, também, extrair informação de documentos comprimidos, ficheiros *html*, imagens, ficheiros de áudio, etc.

O Apache Tika consegue, também, detetar o tipo de ficheiro que está a ser processado, o que simplifica em muito o trabalho de quem está a usar a ferramenta, pelo facto de não se ter de preocupar de como detetar o tipo de ficheiro e qual o *parser* para aquele ficheiro específico.

É importante, ainda, realçar que esta ferramenta é baseada em bibliotecas como a *PDFBox* [11], *Apache POI* [12] ou *Neko HTML* [13], o que indiretamente garante uma grande eficácia e qualidade nos resultados da extração dos metadados. Sendo que nesta tese, o Apache Tika será usado como ferramenta de suporte ao Apache Solr, na medida em que o Tika extrairá a informação dos documentos e depois o Solr indexará a informação relativa a cada ficheiro, de maneira a que posteriormente essa informação possa ser consultada nas pesquisas, de uma forma mais rápida.

2.7 Sumário

Neste capítulo foram introduzidas as noções em que esta tese se baseia, nomeadamente: alguns tipos de ordenação clássicos, ordenação por aprendizagem automática, algumas métricas de avaliação de algoritmos de ordenação e por fim, foram abordadas algumas das tecnologias utilizadas no contexto desta tese, nomeadamente: Apache Solr e Apache Tika.

Estas tecnologias foram importantes no contexto desta tese, pois a tecnologia Apache Solr foi utilizada para indexar os documentos da Quidgest, e no caso da tecnologia Apache Tika, foi utilizada no processo de indexação dos mesmos, para fazer a extração do conteúdo e de algumas propriedades.

Este capítulo serviu essencialmente para estudar o estado da arte e o que já estava feito, pois não faria sentido perder tempo a implementar coisas que já estavam implementas. E este levantamento ajudou bastante, principalmente na extração dos conteúdos dos documentos, já que a tecnologia Apache Tika fazia precisamente isso, pelo que não foi necessário implementar um *parser* próprio. O estudo da tecnologia

Apache Solr foi importante, na medida em que já tinha bastantes coisas de que necessitávamos implementadas, nomeadamente, os *analyzers*, que permitiam aceder aos termos de cada documento e pela ordem em que se encontravam no corpo dos mesmos. Permitia, também, definir os campos que queríamos guardar para cada documento no índice, etc.

O estudo dos vários tipos de ordenação de documentos existentes permitiu comparar todas as abordagens e escolher a melhor delas. Para podermos escolher qual a melhor, foi necessário a utilização de algumas métricas de avaliação e isso foi feito com as métricas estudadas na secção 2.4. O estudo do algoritmo BM25, foi importante na medida em que o Apache Solr ordena os documentos utilizando o algoritmo BM25.

3

Similaridade de documentos

3.1 Contexto e motivação

Como é fácil de perceber, nem todos os documentos são únicos, ou seja, muitos documentos são extensões/cópias de algo já existente. Partindo deste facto, é facilmente percetível um dos principais motivos pelos quais os sistemas de informação, cada vez mais, necessitam de algum tipo de mecanismo de reordenação de resultados sofisticado, para que um utilizador quando faz uma pesquisa, e são retornados os resultados, não veja informação duplicada/similar, pois se existirem vários documentos com um *score* alto mas com informação duplicada/similar, o utilizador irá visualizar a mesma informação em mais do que um documento. Desta forma, não consegue ter acesso a informação mais variada e diversificada, ficando assim limitado, devido ao sistema de reordenação dos resultados não ter em conta o facto de poder existir informação duplicada/similar.

Um documento similar pode ser definido como sendo um documento que contém informação semelhante/duplicada de outro(s). E a semelhança entre documentos pode ser analisada através da semelhança entre os conteúdos dos mesmos, mais concretamente, comparando os seus conteúdos "limpos", ou seja, removendo *stopwords*, fazendo *stemming*, etc.

Estima-se que 30% a 40% das páginas *Web* existentes sejam cópias exatas de outras páginas e, que inclusivamente 2% são cópias íntegras, tendo apenas pequenas diferenças [24]. O crescendo volume deste tipo de dados constitui um dos maiores problemas dos sistemas de informação, pois são armazenados dados duplicados desnecessariamente, aumentando os custos de processamento, bem como os de

armazenamento. Além do mais, os duplicados influenciam negativamente os resultados das pesquisas, como já foi explicado anteriormente.

3.2 Métodos de *hashing* para semelhança em alta dimensão

3.2.1 Locality Sensitive Hashing

Estamos familiarizados essencialmente com algoritmos de *hash* tradicionais, como o MD5 [26] ou SHA [8], que visam criar *hashs* únicos para os dados. Estas funções são construídas para que dados idênticos tenham o mesmo *hash*, assim rapidamente se consegue verificar, por exemplo, se dois documentos são o mesmo ou não, ao contrário do que acontece na família de algoritmos de *locality sensitive hashing*, que caso dois documentos variem pouco, então os seus *hashs* vão ser semelhantes.

Os algoritmos de *locality sensitive hashing* também têm *hashs* únicos, ou seja, dois documentos iguais terão hash's iguais, porém esta abordagem faz com que dados idênticos/similares tenham um *hash* semelhante. Desta forma, poder-se-á verificar se dois documentos são ou não semelhantes, precisando para isso apenas de comparar os seus *hashs*, não existindo necessidade de comparar os conteúdos dos documentos frase a frase, por exemplo.

Ao analisarmos os padrões binários, somos capazes de descobrir algumas informações naturalmente, como a posição de cada bit, o comprimento de uma palavra, a repetição de sequências de bits, etc. Em suma, é uma abordagem mais rápida, eficiente e intuitiva que por exemplo analisar secção a secção de cada par de documentos.

3.2.2 SimHash

Como já foi referido, a verificação da similaridade entre dois documentos é um processo complexo. Para comparar algo, o cérebro humano cria uma lista de critérios e posteriormente, tendo em consideração essa mesma lista, compara cada critério e posteriormente decide se ambos são ou não similares.

O algoritmo *SimHash* procede da mesma maneira: compara os conteúdos de dois documentos, usando como critério a lista de palavras de cada um. Cria uma espécie de impressão digital para cada documento, com o fim de posteriormente essas impressões digitais serem comparadas para verificar a similaridade ou não do referido par de documentos. De seguida enunciamos e descrevemos o algoritmo *SimHash*.

Algoritmo *SimHash*	
1	Definição do tamanho *T* do *hash*
2	Criação de um vetor *V*
3	Para cada palavra:
	Criação de um *hash* com tamanho *T*
4	Para cada palavra:
	Para cada bit *i* do *hash*:
	If(i = 0) V[i]++
	If(i = 1) V[i]--
5	Para cada posição *i* do vetor *V:*
	If(V[i] > 0) i = 1
	If(V[i] < 0) i = 0

No passo 1, é definido o tamanho do *hash* que se pretende utilizar, pois no passo seguinte, a criação do vetor *V* depende desse tamanho, pois esse vetor é para ser criado com o mesmo tamanho, definido no passo 1. No passo 3, para cada palavra presente no conteúdo do documento, é feito o *hash* da mesma. Este *hash* é feito utilizando outras funções de *hash*, tais como: MD5, SHA, MurmurHash [27], etc. O passo 4 é responsável pelo preenchimento do vetor V, onde são percorridos e analisados todos os bits de todos os *hashs*. No passo 5 é construído o *hash* final do documento.

No fim da execução do algoritmo é obtida uma impressão digital do documento, que posteriormente, irá ser comparada com as impressões de outros documentos, com o intuito de se averiguar a sua similaridade. Para isso, é utilizada a distância de *Hamming* [9], que passa por analisar o número de posições de cada par de *hashs* que diferem, ou seja, corresponde ao menor número de substituições necessárias para transformar um *hash* no outro.

A complexidade temporal do algoritmo *SimHash* é igual no melhor, no pior e no caso esperado, na ordem de $O(n * T)$. Sendo que n representa o número de palavras distintas do documento e T representa o tamanho do *hash* de cada palavra. O algoritmo percorre sempre todas palavras e por cada palavra é feito um ciclo que percorre cada bit do *hash* da mesma, por isso a complexidade é igual em todos os casos.

3.3 Documentos similares

Já foi mencionado anteriormente a motivação para a necessidade de serem

descobertos os documentos similares de cada documento presente no topo dos resultados de uma pesquisa, contudo nesta secção vamos explicar mais profundamente esse mesmo contexto e a motivação para essa necessidade e também, explicar de que modo será usado este mecanismo no contexto desta tese.

3.3.1 Motivação

No caso específico da aplicação QSearch, os documentos são muito variados e diversificados, como irá ser descrito na secção 4.1.1, contudo existe também uma grande quantidade de documentos que são cópias/extensões de outros documentos já existentes. Ora quando é realizada uma pesquisa, interessa ao utilizador, sobretudo ter acesso a informação diversificada e que vá ao encontro daquilo que ele procura.

Imaginemos o caso de um utilizador, que pretendia consultar algumas propostas feitas para produtos distintos e, decide fazer uma pesquisa com a palavra "Proposta" e todos os documentos retornados eram propostas, como era suposto, porém haveria uma grande probabilidade de algumas dessas propostas serem cópias/extensões de outras. Desta forma o utilizador iria visualizar propostas semelhantes ou mesmo iguais em alguns casos, o que de facto não era isso que ele pretenderia. Ora por este motivo, foi pensado e posto em prática, no contexto desta tese, o mecanismo de documentos similares, de forma a apresentar o topo dos resultados da pesquisa com informação diversificada, sem documentos similares/duplicados.

Para construirmos o gráfico da figura 9, utilizámos os documentos retornados a 20 pesquisas no QSearch e analisámos os documentos similares em cada lista de documentos retornados. Sendo que na tabela 1 estão apresentadas as pesquisas utilizadas.

Genio	*BSC*	*Propostas Genio*	*Apresentações Mensais Genio*	*QSearch*
Tese Carlos	*Projetos Gestão Documental*	*Ricardo Torres*	*Parceiros Quidgest*	*QWatch*
Phonegap	*QuidSpark*	*Jenkins*	*MVC*	*Procuradoria-Geral República*
Processos jurídicos	*Manipulação historial*	*Outlook addin*	*QWeb3*	*QuiGenio*

Tabela 1. Pesquisas utilizadas para testes do número de documentos similares.

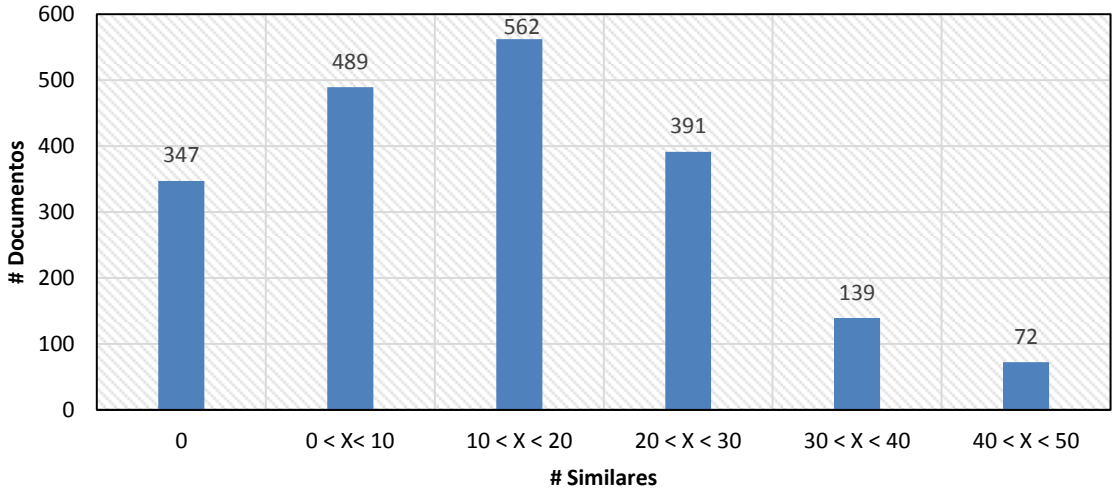

Figura 9. Número de documentos similares nos resultados de 20 pesquisas.

Neste gráfico, no eixo das abscissas estão representados o número de documentos similares e no eixo das ordenadas estão representados o número de documentos. Com a observação do gráfico, é visível o grande número de documentos similares, pois apenas trezentos e quarenta e sete documentos não possuem documentos similares. A grande maioria dos documentos têm em média entre vinte e trinta similares, sendo que desta forma é bastante percetível que os documentos da Quidgest possuem um elevado número de documentos com grande semelhanças e até mesmo pedaços de texto iguais, o que necessariamente, afetará negativamente o topo dos resultados das pesquisas feitas na aplicação QSearch. Sendo que este, é o grande motivo que faz que a deteção de documentos similares seja estritamente necessária no contexto desta tese, com o objetivo de oferecer uma maior diversidade de informação.

3.3.2 Utilização do conceito da similaridade de documentos

Este conceito de documentos similares é utilizado no sistema de reordenação do QSearch através de um algoritmo de diversidade, algoritmo esse que utiliza os *hashs* dos documentos para afinar a diversidade da informação presente nos documentos do topo dos resultados da pesquisa.

Para obter o *hash* de cada documento, é utilizado o algoritmo já acima descrito, *SimHash*. Desta forma, utilizando este algoritmo para obter o *hash*, temos a garantia que o mesmo é computado tendo em atenção o seu conteúdo, a correspondente ordem das palavras, etc. Sendo que assim, é uma forma de documentos similares/duplicados

terem *hashs* semelhantes, que depois são analisados pelo algoritmo de diversidade, com o intuito de remover esses documentos semelhantes/duplicados do topo dos resultados da pesquisa, mesmo tendo estes um *score* alto, porque o que estamos aqui a tentar resolver é a apresentação do topo dos resultados de uma pesquisa com diversidade de informação, contudo tendo em conta o *score* de cada documento.

O cálculo e utilização do *hash* dos documentos é feito em dois momentos distintos no ciclo de utilização da aplicação QSearch, pois o primeiro acontece durante o *DataImport* do SOLR, ou seja, durante a reindexação, ao passo que a segunda ocorre já em tempo de pesquisa.

Optámos por realizar o cálculo do *hash* de cada documento durante o *DataImport* do SOLR, primeiro para diminuir o tempo de execução da reordenação durante o tempo de pesquisa e segundo, constatámos, face à necessidade de encurtar o tempo de execução da reordenação de resultados, que não aumentava consideravelmente o tamanho do índice, visto que para cada documento, apenas acresce o tamanho de um campo do tipo *String*.

Inicialmente, ainda colocámos a hipótese de calcular os documentos similares de cada documento fazendo a análise das frases de cada documento com os restantes, mas depressa compreendemos, com recurso a alguns testes, que era impensável pôr em prática esta ideia, pois aumentaria e muito, o tempo de reindexação do SOLR.

Depois, ainda surgiu uma outra ideia, que passava por escolher as duzentas e cinquenta palavras mais representativas de cada documento (palavras com mais frequência no mesmo), e de seguida realizar uma *query* ao SOLR com estas mesmas palavras. E de seguida, com os documentos retornados pelo SOLR, fazer uma comparação dos *hashs*, contudo esbarrámos no mesmo problema da ideia anterior, pois era um método complexo, na medida em que o número de *hashs* a comparar poderia ser bastante grande.

Por fim, depois de alguma pesquisa, descobrimos a abordagem dos algoritmos de *locality sensitive hashing,* para obter *hashs* que têm em atenção a posição e o conteúdo das palavras, e com recurso a alguns testes e experiências, conseguimos verificar que esta abordagem tinha um menor tempo de execução e produzia bons resultados para o nosso problema específico, como está descrito na secção 3.4.

Na figura 10 encontra-se ilustrado como é feito a indexação do *hash* dos conteúdos documentos, bem como é utilizado, no contexto desta tese, o algoritmo *SimHash*.

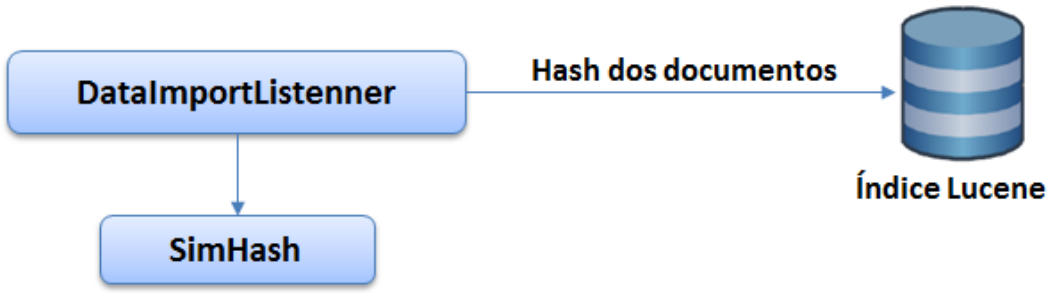

Figura 10. Utilização do algoritmo *SimHash* no processo de *DataImport* do SOLR.

De salientar, que a classe *DataImportListenner* é uma classe que é "despoletada" quando termina o *DataImport* do SOLR, e que posteriormente é responsável por utilizar o algoritmo *SimHash* para criar os *hashs* de cada documento.

É necessário realçar um pormenor bastante importante, pois o algoritmo *SimHash* foi implementado por nós, contudo a ideia não foi pensada por nós, tendo sido aproveitada apenas para o contexto desta tese. Ao passo que o algoritmo de diversidade, uma das principais ideias desta tese, foi totalmente pensado e implementado por nós.

Na figura 11, é possível visualizar um esquema da arquitetura do sistema de reordenação. Sendo que a componente específica onde é utilizado o conceito explicado neste capítulo, a similaridade de documentos, está realçada com a cor vermelha. É já sobre um grupo de documentos relativamente curto, em relação aos documentos iniciais, que o algoritmo de diversidade atua, visto que quem oferece o *input* ao algoritmo é a componente de reordenação por aprendizagem.

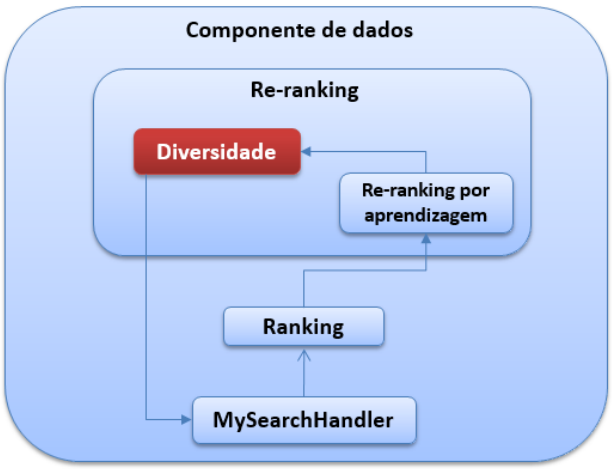

Figura 11. Utilização do conceito de documentos similares.

3.3.3 Criação dos *hashs*

Para serem criados os *hashs* e tendo em conta as características do problema, foi necessário ter em atenção as palavras que eram usadas para a criação dos mesmos, mas também, a posição relativa que cada uma ocupava no corpo do documento. Contudo, antes da criação dos *hashs* dos documentos, através da utilização do algoritmo *SimHash*, foi necessário os documentos passarem por dois processos:

- Remoção de *stopwords,* utilizando para isso uma lista já pré-definida
- Junção das palavras relevantes numa única *String*, separadas por espaços, mantendo a ordem original

Para obter as palavras presentes no conteúdo dos documentos, foi utilizado o *StandardAnalyzer* do SOLR, que por sua vez oferece a possibilidade de percorrer uma estrutura de dados que contém as palavras dos documentos, pela ordem na qual elas se encontram nos mesmos. Este processo é implementado no Solr pelo *tokenStream*. À medida que era percorrido o *tokenStream*, a cada palavra era feita uma análise através de um *Pattern*, para concluir se era ou não uma palavra importante e, caso não fosse, era descartada. Ao passo que quando uma palavra era considerada relevante, ou seja, quando não se encontrava na lista de *stopwords,* é adicionada à *String* onde estavam a ser armazenadas todas as palavras relevantes, separadas por um espaço entre cada uma delas.

Podemos concluir que palavras irrelevantes não foram tomadas em consideração durante a construção dos *hashs* e, mais importante, a ordem pela qual as palavras aparecem no texto é mantida e tida em conta. Desta forma, os *hashs* são mais precisos e têm de facto em conta o conteúdo que realmente interessa, garantindo desta maneira bons resultados no que ao algoritmo de diversidade diz respeito, como poderá ser comprovado no capítulo 5.

3.4 Resultados

Um dos aspetos que requereu a realização de alguns testes foi a definição do limiar de bits em que dois *hashs* poderiam diferir para serem considerados ou não similares. Outro aspeto que foi importante testar e fazer a comparação dos desempenhos, foi a comparação entre a utilização de dois algoritmos diferentes de *locality sensitive hashing* e também a utilização de *hashs* de trinta e dois bits e sessenta e quatro bits, nomeadamente para demonstrar qual dos dois oferece mais precisão e melhores

resultados no que à comparação de *hashs* diz respeito, pois quanta mais precisão o *hash* tiver, mais precisa e segura será a comparação entre os mesmos.

Para a realização dos testes que iremos mostrar, é importante referir que a coleção que utilizámos foi única e exclusivamente os documentos da Quidgest, dado que este era um problema específico da aplicação QSearch. Contudo ainda tentámos realizar estes testes recorrendo para isso à coleção OHSUMED, mas depressa percebemos que era inútil dado o facto de que segundo o algoritmo de similaridade nenhum documento possuía similares na coleção, que depois de analisarmos alguns documentos fez todo o sentido, visto que possuem uma grande variedade de informação e mesmo até ao nível de palavras, não apresentam muitas palavras repetidas entre documentos distintos. Nesta secção iremos mostrar alguns dos testes realizados e resultados obtidos com o intuito de:

- Comparar a precisão dos diferentes tamanhos de *hashs*, utilizando dois algoritmos, com o intuito de perceber qual o tamanho de *hash* mais adequado e qual o melhor algoritmo para o problema em questão
- Definir o limiar do número de bits em que os documentos poderiam diferir para serem considerados similares

Em primeiro lugar importa comparar precisões dos diferentes tamanhos de *hash*, na medida em que quanto mais preciso for o *hash*, mais refinado é o método de similaridade documentos. E em segundo lugar, importa comparar as precisões de dois algoritmos de *hash* diferentes, de modo a obtermos um termo de comparação e podermos escolher o melhor.

Na tabela seguinte, mostramos a diferença entre a precisão de *hashs* de 32 e 64 bits, utilizando os algoritmos *SimHash* e *MurmurHash*. Para a realização destes testes foi utilizado um conjunto de documentos da Quidgest, escolhidos de forma a existirem nessa mesma amostra três grupos de documentos: documentos similares, na medida em que possuíam partes de outros documentos, documentos duplicados e documentos completamente diferentes uns dos outros, sendo que foram utilizados 100 pares de documentos similares e 20 pares de documentos distintos, mas na tabela apenas estão presentes os resultados de 6 pares de documentos similares e distintos.

Os pares de documentos similares foram escolhidos de modo a diferenciarem pouco entre si, de maneira que pudéssemos perceber, por exemplo, qual o peso que um parágrafo tem na construção dos *hashs*. Para isso escolhemos documentos que diferiam pouco entre si, mais concretamente em apenas alguns parágrafos, nomeadamente, para

os pares de documentos apresentados na tabela: o primeiro e terceiro par de documentos diferenciam em dois parágrafos, o segundo par diferencia em apenas um parágrafo, ao passo que o quarto e quinto par diferenciam em três e quatro parágrafos, respetivamente. Os pares de documentos distintos são completamente diferentes, não tendo nenhuma parte igual ou semelhante.

	SimHash		***MurmurHash***	
Tipos de documentos	*32 bits*	*64 bits*	*32 bits*	*64 bits*
Duplicados	0	0	0	0
Duplicados	0	0	0	0
Similares	3	7	2	5
Similares	1	4	3	8
Similares	3	5	5	7
Similares	6	9	6	10
Similares	9	10	10	14
Distintos	23	48	21	50
Distintos	25	55	27	54
Distintos	27	51	28	51
Distintos	18	30	14	29
Distintos	29	58	27	59

Tabela 2. Análise dos *hashs* produzidos pelos algoritmos *SimHash* e *MurmurHash*.

	SimHash		***MurmurHash***	
Tipos de documentos	*32 bits*	*64 bits*	*32 bits*	*64 bits*
Duplicados	0	0	0	0
Similares	4,2	9,4	5,2	8,7
Distintos	25,5	49,8	28,7	48,4

Tabela 3. Média do número de bits diferentes.

Analisando as duas tabelas, no caso dos documentos duplicados não existe diferença para nenhum dos algoritmos, nem para nenhum dos tamanhos de *hash*. As diferenças residem nos documentos similares e distintos, já que o algoritmo *MurmurHash* apresenta uma maior diferença entre os dois tamanhos de *hash*, nomeadamente: para os similares, o algoritmo *MurmurHash* tem uma diferença média menor que o

algoritmo *SimHash*, pelo que concluímos que o algoritmo *SimHash* tem maior precisão a identificar documentos similares.

Perante estes valores, entendemos que o *hash* de 64 bits do algoritmo *SimHash* é o mais preciso, de forma a garantir um maior refinamento no processo de construção dos mesmos, pois como já foi explicado anteriormente, outro ponto importante é a escolha do limiar do número máximo de bits diferentes, para um par de documentos serem considerados ou não similares.

Na tabela 4, estão presentes os resultados dos testes realizados para definir o limiar do número máximo de bits em que os documentos poderiam diferir para serem considerados similares. Os documentos similares diferem pouco entre si, mais concretamente em apenas alguns parágrafos, nomeadamente: o primeiro, segundo e terceiro par de documentos diferenciam em um parágrafo, o quarto, sexto e oitavo par diferenciam em quatro parágrafos, o quinto e sétimo par diferenciam em três parágrafos.

Tipo de documentos	*# Bits diferentes*	*Medida de semelhança do Solr para o 1º documento*	*Medida de semelhança do Solr para o 2º documento*
Duplicados	0	5.4151	5.4151
Duplicados	0	1.6222	1.6222
Similares	2	1.6124	1.5851
Similares	4	3.7500	3.6634
Similares	3	2.8351	2.8193
Similares	12	5.7703	4.8719
Similares	8	5.9099	5.2902
Similares	11	4.9537	3.7552
Similares	6	9.9668	9.2058
Similares	13	3.7372	2.8664
Distintos	43	15.7089	6.6262
Distintos	49	8.3180	2.2391
Distintos	55	5.0909	1.1182
Distintos	42	3.4129	2.5963
Distintos	61	5.4349	0.5249
Distintos	45	4.1499	1.7867

Tabela 4. Número de bits em que os *hashs* dos documentos diferem.

Para cada par de documentos apresentamos o número de bits em que os seus *hashs* diferem, a medida de semelhança dada pelo Solr de cada documento, sendo que esta medida de semelhança revela o quão relevante é o documento para a *query* em questão. Para os documentos similares, as medidas de semelhança são próximas, mostrando que a informação é semelhante. Passando-se exatamente o contrário no que aos documentos distintos diz respeito, as medidas de semelhança são bastante discrepantes, ou seja, os documentos são bastante distintos.

De seguida mostramos uma tabela com a média do número de bits diferentes para os três grupos de documentos para todos os testes realizados, sendo que foram constituídos por: 5 pares de documentos duplicados, 50 pares de documentos similares e 20 pares de documentos distintos.

Tipo de documentos	*Média de bits diferentes*
Duplicados	0
Similares	9,2
Distintos	48,7

Tabela 5. Média do número de bits diferentes para os três grupos de documentos.

Perante estes resultados, em que a média do número de bits diferentes entre documentos similares foi de 9.2, consideramos que definir o limiar de bits para dois documentos serem considerados similares com 15 é um número que vai de encontro aos testes realizados.

Poderíamos ter definido o limiar com 10, visto que a média é de 9.2 bits, mas poderá haver sempre alguns casos em que dois documentos até são similares, mas que os seus *hashs* diferem um pouco mais, como foi possível ver na tabela 4, e atendendo a que a média do número de bits diferentes para os documentos distintos é de 48.7, definindo o limiar com 15 bits não haverá possibilidades de documentos distintos passarem por documentos similares e vice-versa.

3.5 Sumário

Neste capítulo descrevemos o conceito de similaridade de documentos, bem como onde iremos utilizar este conceito no contexto da tese.

Para identificarmos documentos similares, utilizámos a abordagem de *locality*

sensitive hashing, em que a partir da análise aos *hashs* dos documentos, conseguimos identificar documentos similares, mais concretamente a partir do número de bits em que os *hashs* diferirem.

Por último, apresentámos os testes realizados para escolher o algoritmo de *hash*, o número de bits do *hash* e o limiar do número de bits em que os *hashs* de dois documentos poderiam diferir para serem considerados similares.

4

Reordenação por aprendizagem

4.1 Introdução

Reordenação por aprendizagem é um problema de *machine learning*, cujo objetivo é aprender um modelo de ordenação de documentos de acordo com uma dada *query*. Os métodos de reordenação por aprendizagem, permitem que os sistemas de pesquisa possuam uma série de características para realizarem a ordenação dos resultados das pesquisas. Na área de recuperação de informação, a ordenação de documentos de topo é muito importante, na medida em que existe um grande número de candidatos, o que dificulta e torna mais lenta a pesquisa, especialmente em casos, onde os resultados de uma pesquisa podem ascender aos milhões de páginas. Com ordenação por aprendizagem, esperamos conseguir encontrar uma melhor forma de combinar as várias características extraídas dos pares (*query*-documento), usando para isso, exemplos obtidos e verificados anteriormente.

4.1.1 Características dos dados da Quidgest

Os documentos da Quidgest são bastante diversificados, isto é, existem documentos com poucas palavras e consequentemente com um tamanho considerado pequeno, como também existem documentos com mais de quatro mil palavras distintas, com um tamanho bastante considerável.

No que à extração das características de aprendizagem dos documentos diz respeito, por serem documentos com muitas palavras, faz com que esse processo demore mais em tempo de pesquisa, porque primeiramente o SOLR ao retornar os documentos, retorna juntamente as estatísticas de cada um.

Por este ser um processo demorado e complexo, optou-se por uma via alternativa, em que as estatísticas dos documentos são indexadas juntamente com todos os outros campos: título, autores, categorias, etc. Desta forma, quando é feito o pedido ao SOLR, ele já tem as estatísticas indexadas e apenas tem de retornar os documentos com as estatísticas e todos os outros campos pedidos.

No seguinte gráfico está ilustrado o elevado número de palavras distintas que os documentos da Quidgest possuem. Torna-se claro que é um grande obstáculo no processo de extração das *features* dos documentos durante o processo de reordenação dos mesmos, mais concretamente na fase de execução do algoritmo de reordenação automática. E por este motivo tomámos a decisão de indexar as palavras mais representativas de cada documento, de forma a não aumentar exponencialmente o tamanho do índice, nem aumentar o tempo de realização das pesquisas, pois qualquer pesquisa que demore mais do que meio segundo já é considerada pelos utilizadores como sendo lenta e demorada, podendo até, certos utilizadores, pensarem que o sistema bloqueou ou não tem resultados para aquela pesquisa. Para a construção deste gráfico utilizámos todos os documentos presentes no repositório da Quidgest.

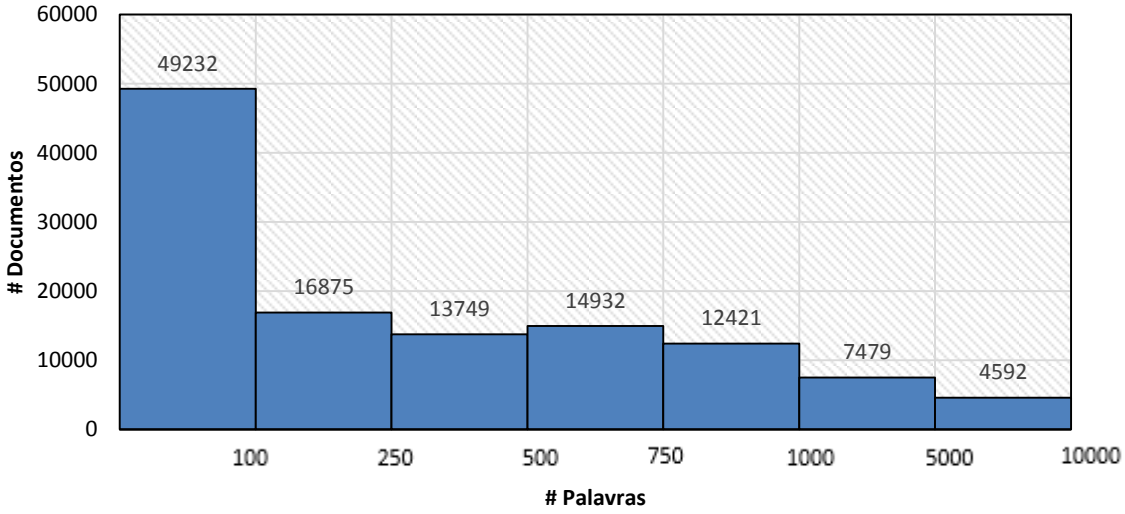

Figura 12. Número de palavras distintas dos documentos da Quidgest.

É notório que os documentos da Quidgest são bastante diversificados, havendo documentos que têm um número relativamente pequeno de palavras distintas, mais concretamente, existem 49232 documentos que têm entre 0 e 100 palavras, 16875 entre 100 e 200 palavras, mas ao mesmo tempo, existem documentos com mais de cinco mil palavras distintas, existem 7479 documentos que têm entre 1000 e 5000 palavras e 4592

que têm entre 5000 e 10000 palavras. Este foi um problema com que tivemos de lidar, principalmente para não termos tempos de pesquisa exagerados.

Outro problema que nos deparámos foi o elevado número de documentos presentes no repositório da Quidgest. Inicialmente implementámos o método para extrair as *features* de todos os documentos presentes nos resultados da pesquisa, contudo existiam pesquisas que tinham mais de cinquenta mil resultados. O método de reordenação através de métodos de aprendizagem automática demorava bastante no processo de extração das características de aprendizagem, na ordem dos 10 segundos, mesmo depois de já estarem a ser indexadas as palavras mais representativas de cada documento.

Tivemos de repensar o método para que o tempo de pesquisa fosse o menor possível e, desta forma, pensámos em restringir o número máximo de documentos que uma pesquisa poderia ter como resultado. Pois, automaticamente, os resultados vindos do SOLR, já vêm ordenados através do algoritmo BM25, o atual estado da arte, ao que aos algoritmos de ordenação diz respeito, já abordado na secção 2. Posto isto, fizemos alguns testes para verificar qual o número máximo de documentos que seriam admissíveis para que os tempos de pesquisa fossem realistas.

Na tabela seguinte mostramos uma média dos tempos de pesquisa, consoante o número de documentos presentes nos resultados da pesquisa. Sendo que estes tempo incluem já a reordenação através do método de aprendizagem automática. Para a construção da tabela foi feita uma média de 10 *queries* para cada caso.

# Documentos	*Tempos de pesquisa (s)*
1000	1,55
750	1,47
500	1,13
250	0,78
100	0,49

Tabela 6. Tempos de pesquisa consoante o número de documentos.

Analisando a tabela, é notório que era impensável incluir no método de reordenação automática todos os documentos retornados pelo SOLR, pois isso implicaria tempos de pesquisa demasiado altos, totalmente intoleráveis em sistemas de pesquisa como este. Fizemos também uma análise ao número de páginas que os utilizadores consultavam nos resultados das pesquisas, e essa estatística está patente no seguinte gráfico.

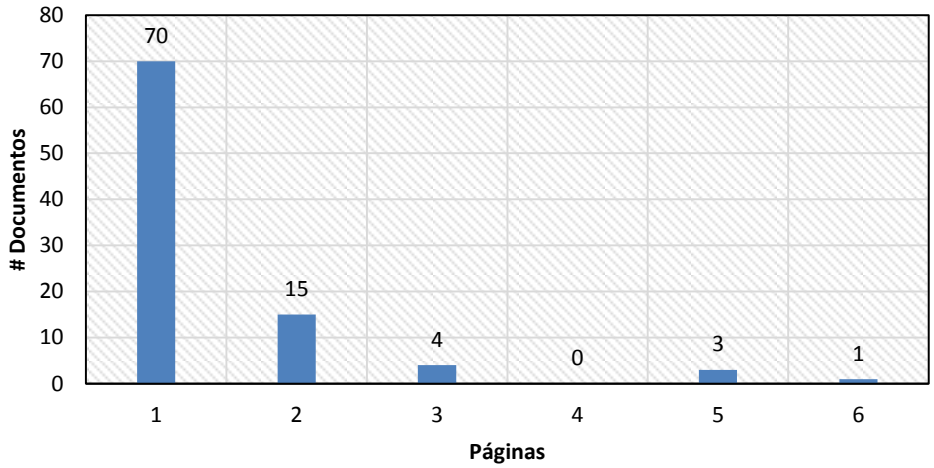

Figura 13. Número de páginas que os utilizadores consultam no QSearch.

Através do gráfico é percetível que a maioria dos utilizadores do QSearch apenas consulta a primeira página dos resultados das pesquisas, o que engloba apenas dez documentos. Contudo existem alguns utilizadores que consultam até à página seis, sendo que até essa página estão contidos sessenta documentos, número esse que vai de encontro do número encontrado através dos testes apresentados na tabela 5.

Perante os testes realizados e estatísticas consultadas, considerámos que limitar o número máximo de documentos retornados pelo SOLR a cem documentos, é um número bastante realista e fundamentado, que vai de encontro ao número de páginas consultadas pelos utilizadores, mas também tem em conta a necessidade de ter tempos de pesquisas curtos. Poderíamos ter limitado o número a sessenta documentos, pois no máximo os utilizadores consultaram até à página 6, contudo poderão existir documentos relevantes mais para trás, e desta forma salvaguardamos documentos que possam estar nessa situação.

4.2 Características de aprendizagem

Os documentos são representados por vetores de características, refletindo a importância dos documentos associados a uma *query*. Na coleção de dados OHSUMED [17], existem 45 características extraídas dos pares (*query*-documento). Existem vários tipos de características disponíveis, mas as que costumam ser mais utilizadas são: a frequência de termos da *query* nos documentos, os outputs do algoritmo BM25, *TF-IDF*,

IDF, etc.

As características, para este tipo de dados, podem ser agrupadas em dois grupos: dependentes do documento (D) e dependentes do documento e da *query* (Q&D). Na tabela seguinte [15], podem ser vistas algumas das características para os dados OHSUMED.

ID	*Descrição da Característica*	*Categoria*
1	$\sum_{q_i \in q \cap d} TF(q_i, d)\ no\ título$	Q&D
2	$\sum_{q_i \in q \cap d} log(TF(q_i, d) + 1)\ no\ título$	Q&D
3	$\sum_{q_i \in q \cap d} log\big(IDF(q_i)\big)\ no\ título$	Q&D
4	$BM25\ no\ título$	Q&D
5	$\sum_{q_i \in q \cap d} TF(q_i, d)\ no\ resumo$	Q&D
6	$\sum_{q_i \in q \cap d} \big(TF(q_i, d)\ .\log\big(IDF(q_i)\big)\big)\ no\ resumo$	Q&D
7	PageRank	D

Tabela 7. Características de aprendizagem dos dados OHSUMED.

4.2.1 Características de aprendizagem adicionais para os dados da Quidgest

Por serem documentos completamente diferentes da coleção de dados OHSUMED, tentámos encontrar mais características específicas para este tipo de documentos, e algumas dessas características específicas encontram-se refletidas na seguinte tabela.

Um aspeto importante é que os documentos da Quidgest não são tão bem estruturados como os da coleção OHSUMED, na medida em que na sua grande maioria apenas possuem campos como: nome, texto do documento, extensão, etc. Não possuindo campos, na sua grande maioria, como título, resumo, etc, ao contrário do que acontece na coleção OHSUMED, que todos os documentos possuem resumo e título. Por esta razão optámos por apenas extrair as características de aprendizagem para o texto do documento e não, também, para o título e resumo, como acontece com os dados OHSUMED.

ID	*Descrição da Característica*	*Categoria*
1	$\sum_{q_i \in q \cap d} TF\_IDF(q_i, d)\ no\ texto$	Q&D
2	$\sum_{q_i \in q \cap d} \log(TF\_IDF(q_i, d) + 1)\ no\ texto$	Q&D
3	$\sum_{q_i \in q \cap d} (\#\ palavras\ /\ \#\ documentos)\ no\ texto$	D
4	# Palavras no texto	D
5	Extensão do documento	D

Tabela 8. Características de aprendizagem dos documentos da Quidgest.

É de salientar que a extensão do documento não é utilizada como característica de uma forma direta, ou seja, a extensão de cada documento é "transformada" num número inteiro, de modo a estar de acordo com as restrições de formato da biblioteca RankLib [3]. Outro facto importante de referir é que no repositório da Quidgest tanto existem documentos com extensão *doc*, *docx*, *xls*, *xlsx*, etc, e para estes casos, o valor atribuído à extensão é o mesmo, ou seja, decidimos não diferenciar ficheiros com extensão *doc* e *docx*.

4.3 Algoritmos de reordenação de documentos

4.3.1 RankNet

O método *RankNet* [5], proposto por Burges (2005), tem uma abordagem *Pairwise*, construindo um modelo de ordenação através do formalismo de Redes Neurais Artificiais, na tentativa de minimizar o número de pares de peritos classificados incorretamente.

A ideia básica do *RankNet* é a utilização de uma rede neural de multicamadas com uma função de custo. Enquanto uma rede neural artificial normal calcula este custo através da medição da diferença entre os valores de saída da rede e os respetivos valores alvo, o algoritmo *RankNet* calcula a função de custo através da medição da diferença entre um par de outputs da rede. Na tentativa de minimizar o valor da função de custo, o algoritmo adapta cada peso na rede de acordo com a inclinação da

função de custo no que diz respeito ao peso.

O algoritmo recebe como parâmetro, o número de iterações que providencia uma rede com *input* e permite atualizar os pesos, *epochs* e, o número de nós que são contidos na camada escondida da rede, *hiddenNodes*. Este último parâmetro, tem um grande impacto no processo de aprendizagem da rede, devido ao facto de que se forem poucos, a rede não conseguirá aprender muitos detalhes. Contudo, se forem muitos nós, a rede não conseguirá prever corretamente, qual o valor de um nó nunca visto anteriormente.

4.3.2 RankBoost

O algoritmo *RankBoost* [6], proposto por Freund (2003), segue a abordagem *Pairwise*, construindo um modelo de ordenação, através do formalismo da abordagem *boosting*, com o intuito de minimizar o número de pares de documentos mal ordenados.

O algoritmo *RankBoost* tem como ideia base treinar um *ranker* fraco em cada iteração e fazer a combinação de todos, obtendo assim uma função final de ordenação. No final de cada iteração, os pares especializados são reajustados, sendo decrementados os pesos dos pares ordenados corretamente e incrementados os pesos dos pares ordenados de uma forma incorreta.

O algoritmo recebe como parâmetro, o número de iterações T e o *threshold*, correspondente ao número de candidatos que devem ser considerados nos *rankers* fracos. Em cada ronda, o algoritmo mantém uma distribuição de pesos sobre as *queries* no conjunto de dados de treino. Quando o algoritmo começa, esses pesos são distribuídos igualmente sobre as todas as *queries*. Em cada iteração, o algoritmo aumenta os pesos dos peritos que não foram ordenados corretamente pelo modelo criado. Como consequência, a aprendizagem da iteração seguinte será focada na criação de *rankers* fracos que podem trabalhar com os especialistas pertencentes às consultas que foram mal ordenadas.

A complexidade temporal do algoritmo *RankBoost* é da ordem $O(T * m * n^2)$, sendo que m representa o número de *queries* de treino e n o número máximo de documentos.

4.3.3 AdaRank

O método *AdaRank*, proposto por Xu (2008), segue a abordagem *Listwise*, construindo um modelo de classificação através do formalismo da abordagem *boosting*, tentando otimizar uma medida de desempenho específica. O processo de aprendizagem do

algoritmo consiste em selecionar repetidamente características de aprendizagem e combiná-los de forma linear. De salientar, que é um algoritmo com um funcionamento muito idêntico ao descrito anteriormente, *RankBoost*.

A ideia básica do *AdaRank* é treinar um *ranker* fraco em cada iteração, e combinar esses *rankers* fracos como sendo a função de ordenação final. Após cada iteração, os especialistas são reajustados, na medida em que é diminuído o peso dos mesmos corretamente ordenados, com base em métricas de avaliação específicas, e aumentando o peso dos que tiveram uma má performance, relativamente à métrica de avaliação.

O algoritmo recebe dois parâmetros, *T* e *E*, o número de iterações que o algoritmo vai executar e uma medida de avaliação de performance, respetivamente. Em cada ronda, o algoritmo *AdaRank* mantém uma distribuição de pesos sobre as *queries* no conjunto de dados de treino. Quando o algoritmo começa, esses pesos são distribuídos igualmente sobre as todas as *queries*. Em cada iteração, o algoritmo aumenta os pesos dos peritos que não foram ordenados corretamente pelo modelo criado. Como consequência, a aprendizagem da iteração seguinte será focada na criação de *rankers* fracos que podem trabalhar com os especialistas pertencentes às consultas que *foram* mal ordenadas. Sendo que a qualidade dos *rankers* é medida através da métrica de desempenho *E*.

O algoritmo *AdaRank* tem uma complexidade temporal da ordem de $O(\ (K+T) * m * n * \log n)$, sendo *k* o número de *features*, *m* o número de *queries* de treino e *n* o número máximo de documentos.

4.3.4 ListNet

O algoritmo *ListNet* [10], otimiza a função de custo *Listwise* com base em uma probabilidade superior, utilizando uma rede neuronal como modelo e o algoritmo *Gradient Descent*, como algoritmo de otimização.

Este método é muito similar ao algoritmo *RankNet*, já descrito anteriormente, sendo que a maior diferença entre ambos, reside no facto de o primeiro usar a lista de documentos como instâncias, ao passo que o segundo utiliza pares de documentos como instâncias. O primeiro, utiliza uma função de custo *Pairwise*, ao passo que o segundo, utiliza uma função de custo *Listwise*. Contudo, quando existem apenas dois documentos para cada *query*, então a função de custo *Listwise*, no algoritmo *ListNet*, torna-se equivalente à função de custo *Pairwise*, no algoritmo *RankNet*.

O algoritmo *RankNet*, tem uma complexidade temporal na ordem de $O(m\ .\ n_{max}^2)$, onde *m* representa o número de *queries* de treino e n_{max}, representa o número máximo de documentos por *query*. No entanto, a complexidade temporal do algoritmo *ListNet* é

da ordem de $O(m\ .\ n_{max})$, e tem resultados mais eficientes.

4.4 Resultados

Nesta secção serão analisados e comparados resultados de reordenação de documentos, utilizando diferentes algoritmos. Algoritmos esses, que já foram descritos na secção anterior e que estão todos implementados na biblioteca de algoritmos de reordenação por aprendizagem automática, RankLib.

Estes testes basearam-se em dois conjuntos de dados: a coleção de dados OHSUMED e os documentos do repositório da Quidgest.

4.4.1 Análise dos dados

OHSUMED

A colecção OHSUMED é constituída por um subconjunto da base de dados MEDLINE, um conjunto de *queries* e juízos de relevância. Os dados são compostos por cerca de 350 mil referências para jornais, cobrindo os anos compreendidos entre 1987 e 1991. Cada referência contém os campos habituais das referências bibliográficas, nomeadamente, título, autor(es) e resumo. O campo destinado aos resumos dos artigos, é truncado, não podendo conter mais do que 250 palavras e cerca de 20%, não possui resumo e/ou autor(es).

Sendo que cada referência contém mais alguns campos, para além dos habituais das referências bibliográficas, como por exemplo, o tipo de publicação, a fonte e os termos atribuídos por humanos, termos esses que foram retirados do vocabulário *MeSH*.

A coleção contém 106 *queries*, geradas por médicos, no decorrer de assistências aos seus pacientes. Adicionalmente, existem juízos de relevância absolutos de documentos, relativamente a cada *query*, gerados por assistentes. Esses juízos de relevância são compostos por três níveis distintos, *n*, *p* e *d*, nomeadamente para representar não relevante, possivelmente relevante e definitivamente relevante, respetivamente.

Quidgest

Os documentos presentes no repositório da Quidgest são bastantes distintos em relação aos da coleção OHSUMED, pois são documentos estruturados e de diferentes tipos, isto é, existem documentos, por exemplo, com extensão *doc*, *pdf*, *txt*, *xls*, etc.

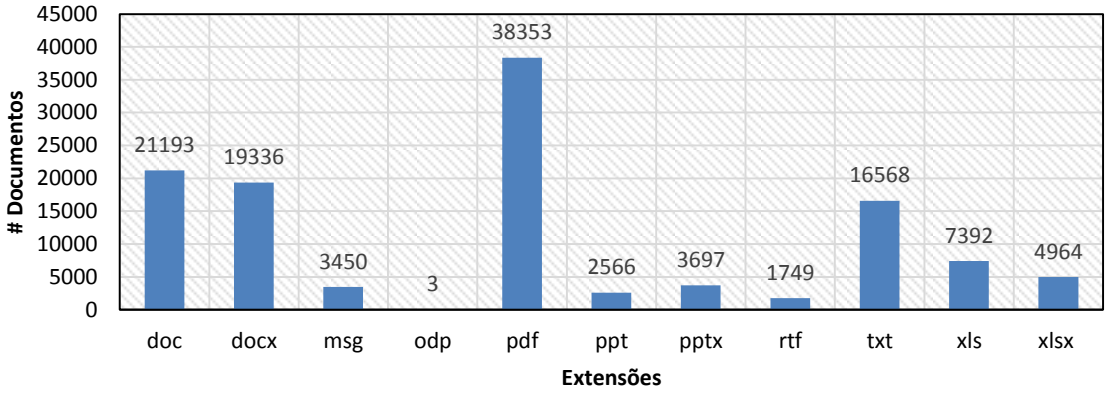

Figura 14. Diferentes tipos de documentos presentes no repositório da Quidgest.

Através deste gráfico, consegue-se perceber a diversidade de documentos no repositório da Quidgest, pois existem onze tipos de documentos indexados. A grande maioria são documentos *pdf*, existindo também uma grande quantidade de documentos *doc* e *docx*.

Atualmente, no repositório encontram-se cerca de cento e vinte mil documentos indexados. Esses documentos pertencem a todas as áreas da empresa, desde Marketing, Gestão Documental, Gestão Financeira, etc. Como já foi referido anteriormente, existem documentos que são cópias integrais ou extensões de outros documentos, o que faz com haja documentos repetidos no próprio repositório.

Ao contrário do que acontece na coleção OHSUMED, os documentos não têm todos os mesmos atributos/campos, pois não existe nenhuma regra implícita na empresa, para que todos os documentos sejam criados com os mesmos atributos/campos. Ora, alguns apresentam data, título, nome do documento, etc, contudo a maioria não contém o atributo título, o que dificultou um pouco na altura de escolha da extração das características de aprendizagem, pois no caso da coleção OHSUMED, puderam ser extraídas características de aprendizagem referentes ao conteúdo, título e à junção do conteúdo e do título, porém no caso dos documentos da Quidgest não podem ser extraídas características referentes a esses campos, pois praticamente nenhum documento apresenta o título como atributo.

4.4.2 Protocolo experimental

A biblioteca RankLib utiliza o formato dos dados LETOR [31], tanto para os ficheiros de dados de treino, tanto para os ficheiros de dados de teste. Este formato é baseado

nas características de aprendizagem descritas na secção 4.2, que foram extraídas, tanto para a coleção OHSUMED, tanto para os documentos da Quidgest. Por este motivo, foi necessário, primeiro, realizar uma análise dos dados, nomeadamente, indexação de cada referência/documento, mas contendo atributos associados; e de seguida, para cada *query* e para cada documento presente nos resultados da *query* em questão, foram calculadas todas as características de aprendizagem para ambas as coleções, mais concretamente quarenta e cinco.

Para realizar estas experiências, os dados e as *queries* foram divididos aleatoriamente em dados e *queries* de teste e de treino, sendo que 90% foi utilizado para treino e os restantes 10% para teste. Usando os dados e as *queries* de treino como *input* para os algoritmos de reordenação por aprendizagem automática, foi realizado o treino, de modo a que fosse aprendido um modelo, para garantir que utilizando aquele modelo, os algoritmos conseguiam ordenar os documentos de uma forma mais correta e eficaz.

Os dados e *queries* de teste foram usados como *input* para os algoritmos de reordenação, para que fossem analisadas as performances dos diversos algoritmos utilizados, com o auxílio das métricas de avaliação abordadas na secção 2.4.

Tendo sido feito *k-fold cross validation* nos testes, com um k igual a 10, ou seja, os dados foram divididos em 10, e no final foi feita a média dos resultados de cada métrica, para cada algoritmo e para cada valor de n (1, 5 e 10).

Os algoritmos utilizados dependem de algumas variáveis, nomeadamente o algoritmo *RankBoost* depende de duas variáveis, o número de rondas para treinar e o *threshold*, sendo que foram utilizados os valores 300 e 10, respetivamente; O algoritmo *ListNet* depende do número de épocas para treinar e do rácio de aprendizagem, tendo sido utilizados os valores 1500 e 0.00001, respetivamente; Os algoritmos *RankNet* e *AdaRank* têm as mesmas variáveis, nomeadamente: o número de épocas para treinar, o número de camadas escondidas, o número de nós escondidos por camada e rácio de aprendizagem, sendo que foram utilizados os valores 100, 1, 10 e 0.0005, respetivamente.

4.4.3 Análise e discussão

Para avaliar os resultados, focámo-nos em duas métricas, *Precision* (*P@n*) e *nDCG* (*nDCG@n*). Estas métricas consideram a relevância dos n resultados de top, ao passo que a métrica *MAP*, considera todos os documentos retornados, por isso não foi utilizada essa métrica.

Coleção de dados OHSUMED

Na tabela seguinte, pode ver-se o resultado dos testes obtidos através da execução de alguns algoritmos de reordenação por aprendizagem, para os dados OHSUMED, implementados pela biblioteca RankLib, com recurso às métricas já referidas. Tendo estas métricas por base as fórmulas apresentadas na secção 2.5.

Os resultados da avaliação de desempenho no retorno de informação do sistema na posição 1, 5 e 10 são apresentados nas figuras 10, 11 e 12. Nestas figuras são comparados os desempenhos de cada algoritmo nas diferentes posições (1, 5 e 10), para cada uma das métricas (*P* e *nDCG*).

	n = 1		*n = 5*		*n = 10*	
	P	*nDCG*	*P*	*nDCG*	*P*	*nDCG*
RankNet	0.500	0.3924	0.4625	0.3537	0.4302	0.3351
AdaRank	**0.604**	**0.4931**	**0.5187**	**0.4622**	**0.476**	**0.3966**
ListNet	0.479	0.4861	0.4604	0.3915	0.4469	0.3607
RankBoost	**0.600**	**0.5333**	**0.56**	**0.4885**	**0.52**	**0.4638**

Tabela 9. Resultados dos algoritmos de reordenação por aprendizagem para a coleção de dados OHSUMED.

Os seguintes gráficos apresentam as performances dos vários algoritmos relativamente às duas métricas utilizadas, para cada valor de *n* analisado.

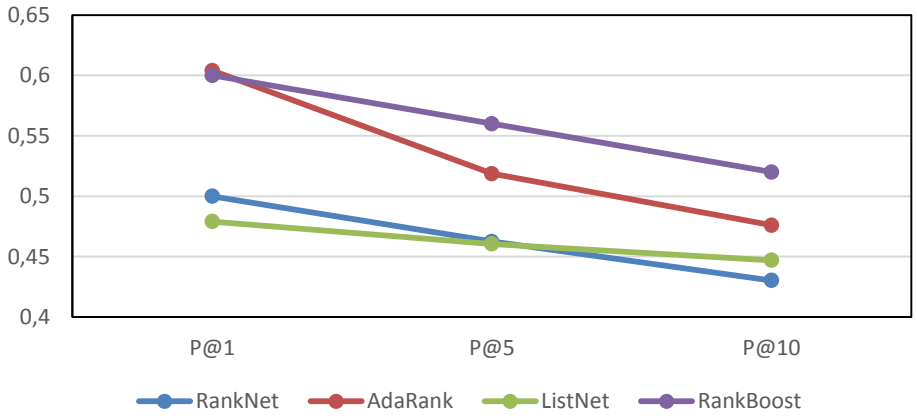

Figura 15. Resultados para a métrica *Precision* para a coleção OHSUMED.

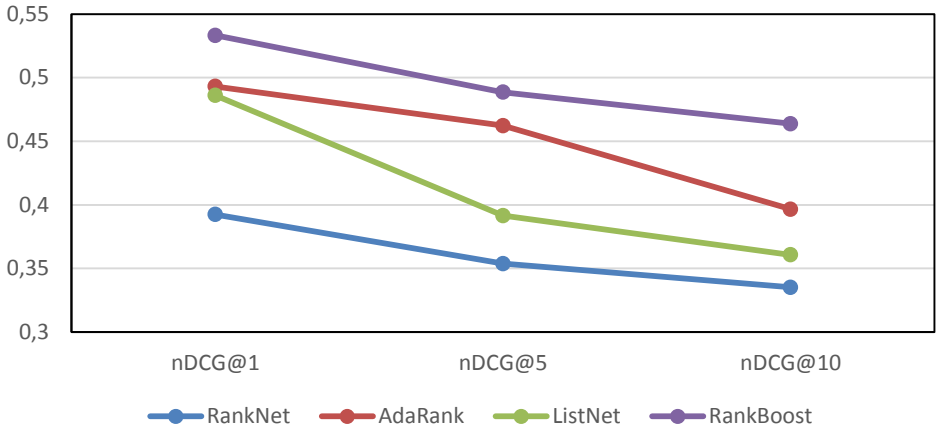

Figura 16. Resultados para a métrica *nDCG* para a coleção OHSUMED.

Analisando os gráficos, constatamos que todos os algoritmos têm uma melhor performance quando é considerado apenas o primeiro resultado do topo dos documentos retornados. Nos testes realizados, os dois melhores algoritmos em todos os casos são o *AdaRank* e o *RankBoost*. Importa analisar o que será mais vantajoso: melhor *Precision* ou melhor *nDCG*. Ambas as métricas já foram descritas na secção 2.5, mas importa referir que a métrica *Precision* é calculada através da soma da relevância atribuída manualmente aos *n* melhores documentos. Porém, o valor da métrica não se altera, se por exemplo, o documento mais relevante passar do primeiro para o quinto lugar. Ao passo que a métrica *nDCG* mede a utilidade/ganho dos *n* primeiros documentos com base na sua posição na lista de resultados. O que quer dizer que para o nosso caso específico, ter melhores resultados no que à métrica *nDCG* diz respeito, é mais importante, do que ter melhor valores de *Precision*.

Se tivéssemos só em atenção o valor da métrica *nDCG*, o melhor algoritmo seria o *RankBoost*, contudo podemos fazer uma análise, por exemplo, em termos de custos computacionais, ou seja, comparar as complexidades temporais dos dois melhores algoritmos. O algoritmo *AdaRank* tem uma complexidade temporal da ordem de $O((K+T)*m*n*\log n)$, e o algoritmo RankBoost tem uma complexidade temporal da ordem de $O(T*m*n^2)$, pelo que o algoritmo AdaRank tem uma menor complexidade temporal em comparação com o algoritmo *RankBoost*. Poderá ser vantajoso em casos em que se necessite de otimizar mais a rapidez do processo de reordenação a utilização do algoritmo *AdaRank*, em detrimento do algoritmo *RankBoost*.

Coleção de dados da Quidgest

Para realizar os testes com os documentos da Quidgest deparámo-nos com a ausência

de dados de treino. Pois neste caso não possuíamos informação predefinida relativamente à importância dos documentos para as pesquisas efetuadas pelos utilizadores, como aconteceu para a coleção OHSUMED.

Para resolver esta questão, utilizámos a informação relativa aos ficheiros acedidos por cada utilizador e, desta forma, obtivemos informação relativa à importância dos documentos, que são realmente relevantes para cada pesquisa feita no QSearch, de modo a criar um modelo de aprendizagem, para ser utilizado pelos algoritmos de reordenação. Cada vez que um ficheiro era acedido por um utilizador, era armazenado numa base de dados, a *query* e o *id* do ficheiro acedido. Para a contabilização dos ficheiros acedidos, utilizámos 1457 pesquisas feitas ao QSearch, sendo que apenas 727 são distintas.

Na tabela seguinte encontram-se espelhados os testes realizados com um conjunto de documentos da Quidgest. Foram utilizados os mesmos algoritmos e as mesmas métricas que nos testes realizados com a coleção de dados OHSUMED.

	n = 1		*n = 5*		*n = 10*	
	P	*nDCG*	*P*	*nDCG*	*P*	*nDCG*
RankNet	0.3880	0.3490	0.3408	0.3113	0.3198	0.2899
AdaRank	**0.4590**	**0.4092**	**0.4150**	**0.3798**	**0.3803**	**0.3441**
ListNet	0.3923	0.3729	0.3582	0.3312	0.3299	0.3091
RankBoost	**0.4732**	**0.4309**	**0.4238**	**0.4031**	**0.3921**	**0.3619**

Tabela 10. Resultados dos algoritmos de reordenação por aprendizagem para documentos da Quidgest.

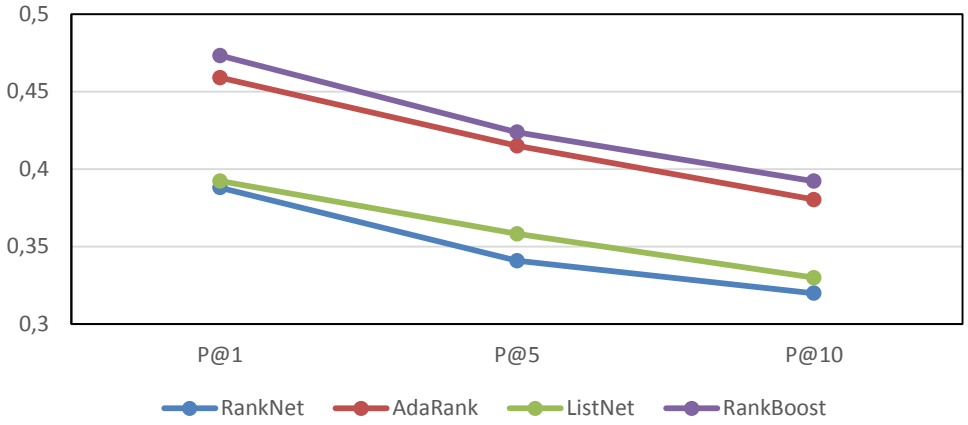

Figura 17. Resultados para a métrica *Precision* para a coleção da Quidgest.

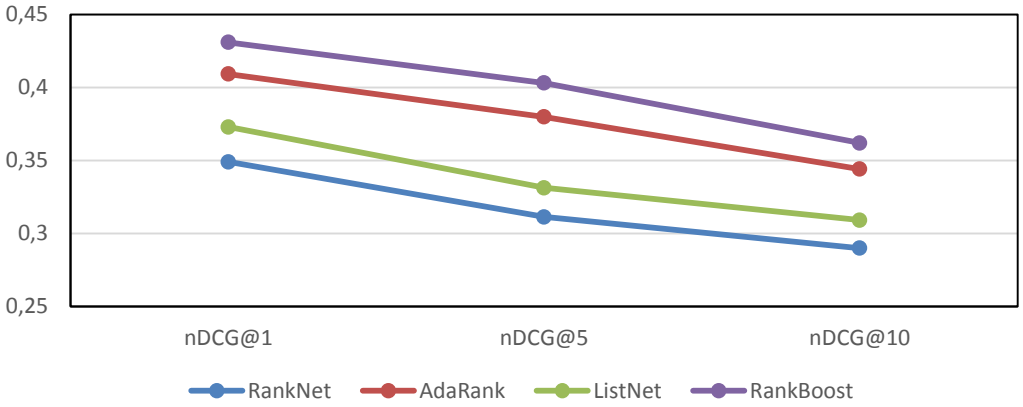

Figura 18. Resultados para a métrica *nDCG* para a coleção da Quidgest.

Todos os algoritmos apresentam melhor performance relativamente às duas métricas, quando apenas é considerado o primeiro elemento do topo dos resultados. Para o caso específico dos documentos da Quidgest importa ter um melhor desempenho para a métrica *nDCG*, para o top 10 dos resultados das pesquisas, pois como já foi explicado anteriormente, a grande maioria dos utilizadores apenas consulta a 1ª página dos resultados das pesquisas. Nesse caso, o melhor algoritmo é o *RankBoost*. Contudo o algoritmo *AdaRank* tem valores próximos do *RankBoost*, e como a complexidade temporal do *AdaRank* é menor que a do *RankBoost*, poderá haver casos em que seja mais propícia a utilização do algoritmo *AdaRank*.

4.5 Sumário

Neste capítulo apresentámos alguns algoritmos de reordenação por aprendizagem, bem como algumas características de aprendizagem da coleção OHSUMED e dos documentos da Quidgest.

Através das experiências com a coleção OHSUMED e com os documentos da Quidgest concluímos que para ambos os casos, o algoritmo *RankBoost* apresentava melhores resultados, contudo o algoritmo *AdaRank* não está muito longe do *RankBoost*. Como a complexidade temporal do *AdaRank* é mais baixa que a do *RankBoost*, poderá haver casos em que seja mais vantajoso usar o *AdaRank*.

5

Diversidade

5.1 Introdução

Para uma determinada consulta, um sistema de informação deve retornar uma lista ordenada, que respeite quer a amplitude da informação disponível, quer qualquer ambiguidade inerente à pesquisa. A palavra "jaguar" representa um exemplo de uma pesquisa ambígua. Em resposta a esta pesquisa, um sistema de informação pode retornar uma mistura de documentos que têm informação relativa a carros e animais. Estes documentos fornecem um quadro completo de todas as interpretações possíveis para esta palavra.

O ideal é que a ordenação dos documentos tenha em consideração o sentido da palavra para o utilizador. Se o utilizador fizer muitas pesquisas sobre carros, então talvez seja apropriado apresentar os documentos relativos a carros no topo dos resultados da pesquisa e depois aparecerem os documentos relativos a animais. Em primeiro lugar, devem constar documentos que cubram os aspetos-chave de cada tópico, ao passo que depois no seguimento da lista, devem constar os restantes documentos, de forma a evitar informação similar no topo dos resultados da pesquisa.

A criação de um sistema de informação que tem em conta a similaridade entre documentos, ou seja, a diversidade de informação apresentada pelos mesmos apresenta muitos desafios. Um desses desafios é a falta de uma forma clara e objetiva de definir se o resultado para uma determinada pesquisa é bom ou não em termos da diversidade de informação. As métricas de avaliação geralmente utilizadas - como *Precision* e *nDCG*, supõem que a relevância de cada documento pode ser avaliada de forma isolada, independentemente de outros documentos. Ajustar sistemas de

informação de forma a otimizar as suas performances em relações às métricas já citadas, pode levar à existência de resultados insatisfatórios no que à diversidade de informação respeito.

A solução de excluir documentos similares da coleção, não resolve o problema, pois apenas o esconde. Em vez disso, o sistema de informação deve ser capaz de lidar com documentos similares e reordenar os documentos de acordo com a diversidade de informação que os mesmos apresentam, beneficiando documentos que contêm uma maior diversidade de informação, ou seja, documentos que contêm informação sobre determinados tópicos e não são similares de nenhum outro documento, ao invés de possuir informação que já está contida noutro(s) documento(s).

A diversidade pode definir-se como sendo a qualidade de um sistema de informação conseguir evitar redundância, ou seja, quando apresenta ao utilizador dois documentos com conteúdo igual ou semelhante, é óbvio que um desses documentos acrescenta pouca utilidade para o utilizador, na medida em que está a tirar a possibilidade de outro(s) documento(s) estarem no topo dos resultados e desta forma o utilizador pode nem sequer visualizar esses documentos que estão mais para baixo na ordenação.

Figura 19. Reordenação sem e com diversidade.

Nesta figura 19 está ilustrado o principal objetivo desta tese, reordenar o topo dos resultados das pesquisas segundo a diversidade de informação apresentada. Neste exemplo, vamos assumir que blocos que tenham a mesma cor, significa que são documentos que abordam o(s) mesmo(s) tópico(s). Ora no lado esquerdo da imagem, os documentos estão ordenados, não tendo em consideração a cor dos blocos, o que já não acontece no lado direito, onde blocos com cores iguais não aparecem

consecutivamente. Do lado esquerdo, está a reordenação feita pelo algoritmo de aprendizagem e do lado direito está a reordenação feita pelo algoritmo de diversidade. É percetível que do lado direito da figura, existe uma maior diversidade de cores no topo da reordenação, ou seja, existe uma maior variedade de documentos que abordam temas diferentes, ao contrário do que acontece no lado esquerdo, que existem cores iguais seguidas na reordenação, o que quer dizer que existe menos diversidade nos documentos de topo.

Supondo que numa coleção existe um número considerável de documentos similares, e caso seja feita uma pesquisa em que um desses documentos similares é relevante, então todos os outros irão estar presentes no topo dos resultados da pesquisa. No caso dos documentos da Quidgest, existem muitos documentos similares, como já foi apresentado na secção 3.3, pelo que era estritamente necessário, de forma a aumentar a diversidade de informação no topo dos resultados das pesquisas, desenvolver um mecanismo para reordenar o topo dos resultados das pesquisas segundo a sua diversidade de informação, pelo que este foi um dos maiores propósitos desta tese e vai ser explicado na secção 5.3 como foi desenvolvido esse mecanismo.

5.2 Medidas de diversidade

Como foi mencionado anteriormente, avaliar a diversidade de informação nos resultados de uma pesquisa não é um tema consensual, por ser algo subjetivo e esse facto impossibilita o uso das métricas de avaliação usuais, ao contrário do que acontece quando, por exemplo, queremos medir a eficácia de algoritmos a reordenar documentos segundo a sua importância para as pesquisas, independentemente da diversidade de informação.

Para avaliar a diversidade de informação no topo dos resultados das pesquisas, de uma forma mais formal, sem qualquer tipo de subjetividade, existe uma medida de avaliação baseada num novo conceito, designado por *information nuggets*. Um *information nugget* representa um tópico de informação que um documento aborda, podemos considerar que um conjunto de *information nuggets* representam uma necessidade de informação. Concetualmente, um *information nugget* pode representar um facto específico relacionado com uma necessidade de informação sobre um tópico específico ou até mesmo uma resposta a uma determinada pergunta. Pode, eventualmente, representar uma exigência estrutural, indicando que um documento pertence a uma coleção específica, tendo sido escrito num determinado período de

tempo, ou até que a sua localização se encontra em uma página web específica.

São atribuídos aos documentos valores de relevância, com base no número de *information nuggets* que cada um contém, ou seja, quanto mais *information nuggets* um documento possuir, maior será o seu grau de relevância. Além disso, dada uma lista ordenada de documentos, se um *information nugget* ocorrer num documento, então o seu valor em outros documentos deverá ser menor, favorecendo desta forma documentos com tópicos diferentes, ou seja, favorecendo quem apresentar maior diversidade de informação. Com estes valores de relevância disponíveis, uma medida de avaliação como *nDCG* já poderia ser aplicada para medir a eficácia dos resultados. A extensão da métrica *nDCG* [23], para avaliar os valores de relevância com base em *information nuggets*, é conhecida como *α-nDCG*.

Mais formalmente, podemos modelar a necessidade de informação dos utilizadores como sendo um conjunto de *information nuggets* $\{n_1, \dots, n_m\}$. Dada uma lista ordenada de documentos $\langle d_1, d_2, \dots \rangle$, $N(d_i, n_j) = 1$ se o documento d_i possuir o *information nugget* n_j, caso contrário, $N(d_i, n_j) = 0$. O número de *information nuggets* contidos no documento d_i é dado pela seguinte fórmula:

$$\sum_{j=1}^{m} N(d_i, n_j) \tag{15}$$

Se a diversidade de informação não fosse um problema, o número de *information nuggets* que cada documento contém poderia ser usado diretamente como um valor de relevância para os mesmos. Contudo, há pequenos ajustes que têm de ser feitos, devido ao facto de poderem existir *information nuggets* repetidos, sendo necessário ter em consideração o menor valor da informação que seja similar, ou seja, documentos que têm os mesmos *information nuggets* devem ter uma relevância menor em relação a documentos que têm *information nuggets* distintos, visto que um dos maiores propósitos desta tese é aumentar a diversidade de informação presente no topo dos resultados das pesquisas. Para fazer este ajuste, primeiro foi definido:

$$r_{j,k-1} = \sum_{i=1}^{k-1} N(d_i, n_j) \tag{16}$$

Sendo que $r_{j,k-1}$ representa a relevância do *rank* de 1 a *k*-1 para *o information nugget j* e, por conveniência, foi definido $r_{j,0} = 0$. O *k* elemento do vetor de ganhos *G* foi definido

através da seguinte equação:

$$G[k] = \sum_{j=1}^{m} N(d_k, n_j) \times \alpha^{r_{j,k-1}} \quad (17)$$

Onde α é uma constante, que varia entre zero e um, representando a redução do valor ganho de um *information nugget* repetido. Na equação 17, o fator $\alpha^{r_{j,k-1}}$ é o responsável pela desvalorização dos *information nuggets* repetidos, isto é, este fator influencia negativamente o valor de relevância dos documentos que contiverem os mesmos *information nuggets*.

Na tabela seguinte mostramos um exemplo de alguns documentos e os respetivos tópicos que cada um aborda, assinalando para isso com uma cruz os tópicos de cada documento [23]. Esta tabela tem o objetivo de mostrar a ordenação dos documentos usando apenas o número de *information nuggets* de cada documento e realçar a importância que os *information nuggets* repetidos têm, devido ao facto de retirarem do topo dos resultados documentos com *information nuggets* distintos.

Documentos	*1*	*2*	*3*	*4*	*5*	*6*	*Total*
A	-	X	-	X	-	-	2
B	-	X	-	-	-	-	1
C	-	X	-	-	-	-	1
D	-	-	-	-	-	-	0
E	X	-	-	-	-	X	2
F	X	-	-	-	-	-	1
G	-	-	X	-	-	-	1
H	X	-	-	-	-	-	1
I	-	-	-	-	-	-	0
J	-	-	-	-	-	-	0

Tabela 11. Documentos com os respetivos tópicos abordados. [23]

Se a ordenação fosse feita apenas pelo número de *information nuggets* de cada documento, então a ordenação final seria: A, E, B, C, F, G, H, D, I, J. Porém, por exemplo, o documento B possui um dos *information nuggets* do documento A, o que quer dizer que aborda o mesmo tópico, ou seja, ambos têm informação similar Acontecendo a mesma situação com o documento C em relação aos documentos A e B.

Logo por estes dois exemplos conseguimos perceber que a diversidade de informação no topo dos resultados é pequena, pois dos 4 primeiros documentos, apenas 2 apresentam *information nuggets* diferentes. Se analisarmos os *information nuggets* do documento G, constatamos que é o único que tem informação sobre o terceiro tópico e está em 6º na ordenação por número de *information nuggets*, por apenas ter um *information nugget*, mas é o único documento que possui aquele *information nugget*, algo que deveria ser mais valorizado.

Perante esta ordenação conseguimos perceber que mesmo assim existem documentos que abordam temas iguais no topo da ordenação, portanto esta abordagem ainda não resolve o problema da diversidade de informação por completo. Esse problema é resolvido com a utilização das equações 16 e 17, que já têm em conta o número de vezes que um *information nugget* ocorre.

Posto isto, tendo em conta a diversidade de informação, ou seja, o número de *information nuggets* repetidos, e de acordo com a equação 17, utilizando um valor de α = 0.5, obtemos o seguinte vetor de ganhos *G*:

Documentos	*1*	*2*	*3*	*4*	*5*	*6*	*Total*	*G*
A	-	X	-	X	-	-	2	2
B	-	X	-	-	-	-	1	1/2
C	-	X	-	-	-	-	1	1/4
D	-	-	-	-	-	-	0	0
E	X	-	-	-	-	X	2	2
F	X	-	-	-	-	-	1	1/2
G	-	-	X	-	-	-	1	1
H	X	-	-	-	-	-	1	1/4
I	-	-	-	-	-	-	0	0
J	-	-	-	-	-	-	0	0

Tabela 12. Vetor de ganhos considerando *information nuggets* repetidos.

Perante estes valores, na tabela seguinte estão as duas ordenações finais, considerando e não considerando os *information nuggets* repetidos:

Não considerando os nuggets repetidos	*A*	*E*	*B*	*C*	*F*	*G*	*H*	*D*	*I*	*J*
Considerando os nuggets repetidos	*A*	*E*	*G*	*B*	*F*	*C*	*H*	*D*	*I*	*J*

Tabela 13. Comparação entre as ordenações que consideram e não consideram *information nuggets* repetidos.

Comparando as duas ordenações, é visível a penalização dos documentos que possuem *information nuggets* repetidos. De realçar a valorização do documento G, que sobe da sexta para a terceira posição na ordenação, que já tem em consideração a existência de *information nuggets* repetidos. Portanto, a abordagem de considerar os *information nuggets* repetidos é mais realista e reflete melhor a importância da diversidade de informação, indo ao encontro do que necessitamos no contexto desta tese.

5.3 Algoritmo de diversidade

Nesta secção iremos descrever o algoritmo desenvolvido no contexto desta tese, que tem como principal objetivo aumentar a diversidade de informação no topo dos resultados das pesquisas, efetuadas ao sistema QSearch. Na base do algoritmo estão os *hashs* abordados no capítulo 3, na medida em que a similaridade dos documentos é comparada através dos *hashs* dos mesmos. O algoritmo trabalha sobre uma lista de documentos já reordenada pelo algoritmo de reordenação por aprendizagem automática, ou seja, o *input* do algoritmo de diversidade é o *output* do algoritmo de reordenação por aprendizagem automática.

Durante o desenvolvimento do algoritmo pensámos em várias abordagens, mas esbarrámos sempre nos problemas temporais, isto é, encontrámos outras abordagens que até apresentavam bons resultados em termos de similaridade, mas a complexidade temporal do algoritmo seguindo essas abordagens era demasiado pesada e necessitávamos de um algoritmo que tivesse bons resultados na similaridade dos documentos, mas também que fosse rápido, pois este algoritmo seria executado em tempo de pesquisa.

Na implementação do algoritmo é mantida uma lista de documentos principal, que guarda os documentos já reordenados. Essa estrutura de dados é necessária na medida em que o algoritmo é um algoritmo recursivo e em cada execução do mesmo, são guardados numa estrutura de dados secundária, os documentos que têm documentos similares na lista de documentos principal. No fim de cada execução do algoritmo é

feita uma chamada recursiva para reordenar essa lista de documentos secundária.

Algoritmo de Diversidade 1

***Input*:** Conjunto de documentos *D*, parâmetro *N*
***Output*:** Conjunto de documentos *T*
T = []
Procedimento: Diversidade

```
1   If( T.size() == N ) retornar T
2   S = []
3   Para cada documento d ∈ D:
4       sim = false
5       Para cada documento i ∈ T:
6           If( Distancia_Hamming( hash_d , hash_i ) ≤ 15 )
7               S.add( d )
8               sim = true
9               break
10      If( sim == false ) T.add( d )
11  If( S.size() > 0 ) Diversidade(S, N)
```

O algoritmo sendo recursivo possui uma condição de paragem, como se pode visualizar na linha 1, em que caso o conjunto de documentos *T*, que representa os documentos do topo dos resultados da pesquisa, que já se encontram ordenados segundo a sua diversidade, contenha já os *N* elementos, então o algoritmo retorna os documentos do conjunto *T*.

Na linha 2 é inicializada uma estrutura de dados *S*, que guarda os documentos que já contêm documentos similares no conjunto *T*, que posteriormente irão ser reordenados com uma chamada recursiva ao algoritmo. Cada documento presente no conjunto *D* é comparado com os que já estão no conjunto *T*, com o intuito de impedir que documentos similares fiquem no topo dos resultados da pesquisa, como é ilustrado a partir do passo 3 do algoritmo. Cada par de documentos são comparados através dos seus *hashs*, sendo que os *hashs* são comparados através da distância de *Hamming*, que se limita a comparar o número de bits diferentes entre os dois *hashs*. Caso os *hashs* dos dois documentos difiram em apenas 15 ou menos bits então é porque são similares, e nesse caso o documento *d* é adicionado ao conjunto *S*, é atribuído o valor *true* à variável *sim*, para que no fim do ciclo se saiba que foram encontrados documentos

similares no topo dos resultados e é feito um *break* para terminar o ciclo, pois neste caso existe pelo menos um documento similar no topo dos resultados. Caso a variável *sim*, no fim do ciclo, for igual a *false*, ou seja, se não foram encontrados documentos similares no topo dos resultados daquele documento, então o documento *d* é adicionado ao topo dos resultados.

No fim da execução do algoritmo, caso o conjunto *S* não esteja vazio, é necessário reordenar esses documentos, então é feita uma chamada recursiva, passando como argumento o conjunto *S* e o parâmetro *N*, como está ilustrado na linha 11.

Na tabela seguinte analisamos a complexidade temporal do algoritmo de diversidade no melhor, no pior e no caso esperado. Sendo que *n* representa o número total de documentos que o algoritmo de diversidade tem de reordenar, ou seja, o número de elementos presentes no conjunto *D*, na primeira chamada do algoritmo.

	Complexidade Temporal
Melhor Caso	$O(n^2)$
Pior Caso	$O(n^3)$
Caso Esperado	$O(n^3)$

Tabela 14. Análise da complexidade temporal do algoritmo de diversidade 1.

O melhor caso acontece quando nos documentos não existe nenhum documento similar, portanto não é feita nenhuma chamada recursiva, ao passo que o pior caso acontece quando todos os documentos têm pelo menos um similar na lista de documentos. Contudo, o caso esperado é que sejam feitas no máximo $n/2$ chamadas recursivas, logo a complexidade temporal será $O(n^2 \times n/2)$, que é da ordem de complexidade de $O(n^3)$.

Para não nos restringirmos a uma única solução e podermos ter uma medida de comparação, decidimos implementar outro algoritmo de diversidade, baseado de igual forma nos *hashs* dos documentos, que se encontra descrito de seguida.

Algoritmo de Diversidade 2

***Input*:** Conjunto de documentos *D*

***Output*:** Conjunto de documentos *T*

Procedimento: Diversidade

```
1   PQ = new PriorityQueue()
2   T = []
3   Para cada documento d ∈ D:
4       Para cada documento i ∈ D:
5           If( d != i )
6               PQ.add( d, i, Distancia_Hamming( hash_d , hash_i ) )
7   Para cada tripo t ∈ PQ:
8       If( !T.contains( t.first ) )
9           T.add( t.first )
10      If( !T.contains( t.second ) )
11          T.add( t.second )
```

O conjunto *PQ* é uma fila com prioridade, sendo que os triplos são ordenados pela distância de *Hamming*, ou seja, os pares de documentos que tiverem maiores distâncias de *Hamming* estarão no topo da fila. A ideia deste algoritmo está em colocar no topo dos resultados, os documentos que apresentarem maiores distâncias entre eles, ou seja, documentos distintos aparecerão no topo dos resultados, de maneira a que a diversidade de informação no topo seja maior.

Na tabela seguinte analisamos a complexidade temporal do algoritmo de diversidade no melhor, no pior e no caso esperado. Sendo que n representa o número total de documentos que o algoritmo de diversidade tem de reordenar, ou seja, o número de elementos presentes no conjunto *D*, na primeira chamada do algoritmo.

	Complexidade Temporal
Melhor Caso	$O(n^2)$
Pior Caso	$O(n^2)$
Caso Esperado	$O(n^2)$

Tabela 15. Análise da complexidade temporal do algoritmo de diversidade 2.

Todos os casos deste algoritmo têm complexidade $O(n^2)$, pois o algoritmo irá

sempre percorrer o conjunto de documentos em dois ciclos, qualquer que seja o número de documentos do conjunto a reordenar. Em comparação ao outro algoritmo desenvolvido, a complexidade deste são menores, sendo quadrático em todos os casos, ao passo que o outro está na ordem de $O(n^3)$, no pior e no caso esperado. Em resumo, em termos de complexidade este algoritmo é melhor que o outro.

5.4 Resultados

Para avaliar a eficiência dos algoritmos de diversidade, desenvolvidos no contexto desta tese, não possuímos informação suficiente para os poder avaliar com recurso às métricas apresentadas na secção anterior, nomeadamente, não possuímos informação suficiente relativa aos tópicos que cada documento do repositório da Quidgest aborda. Ao contrário do que aconteceu, por exemplo, nos testes realizados no capítulo 4, em que as performances dos algoritmos de reordenação por aprendizagem foram avaliadas com base em duas métricas.

Neste caso o que realmente interessa é a eficiência do mesmo, em reordenar o topo dos resultados das pesquisas segundo a diversidade de informação dos mesmos, e esta eficiência, sem informação relativa aos tópicos de cada documento, não pode ser avaliada com base em métricas, é algo que exige uma avaliação subjetiva. Por este motivo, os testes para avaliar a performance do algoritmo de diversidade foram mais complicados.

Perante as condicionantes já explicadas, resolvemos fazer verificação manual dos tópicos de informação presentes nos resultados de pesquisas efetuadas a uma amostra de 200 documentos pertencentes ao repositório da Quidgest, onde existiam documentos similares, duplicados e distintos. Para a realização destes testes etiquetámos cada documento com um ou mais tópicos e no fim da reordenação verificámos quantos tópicos distintos e repetidos constavam no topo dos resultados.

Na escolha dos documentos, tivemos em atenção a informação contida por cada documento, pois necessitávamos de documentos que abordassem os mesmos temas, mas também necessitávamos de ter documentos que abordassem temas completamente diferentes, de forma a garantir que o algoritmo ia trabalhar sobre uma coleção de documentos em que existiam diferentes temas

Sendo que os utilizadores praticamente só acedem aos documentos da primeira página dos resultados, os nossos testes para avaliar a diversidade da informação incidiram sobre os dez documentos de topo dos resultados das pesquisas. Utilizámos

20 *queries* que os utilizadores do QSearch já tinham feito no sistema, tendo sindo escolhidas de maneira a que conseguíssemos ter documentos relevantes e não relevantes para essas pesquisas.

Na tabela seguinte estão presentes os tópicos utilizados e quantos documentos abordam cada tópico, sendo que estes tópicos já estavam definidos no sistema QSearch.

Tópicos	*# Documentos*
Genio	23
BSC	43
FCT	9
Apresentações	25
QSearch	12
SINGAP	18
Património	20
Gestão Documental	30
Gestão Financeira	20
Quidgest Timor	18
Quidgest Macau	22
Phonegap	13
QuidSpark	12
Jenkins	7
MVC	28

Tabela 16. Número de documentos por tópico.

Nas três tabelas seguintes estão os resultados dos testes realizados aos tópicos abordados pelos documentos presentes no top 10 das respostas às pesquisas, para o algoritmo BM25 e os dois algoritmos de diversidade desenvolvidos. Para cada caso em cada algoritmo utilizámos as mesmas 20 *queries*.

Algoritmo	*Média tópicos distintos@10*	*Média tópicos repetidos @10*
BM25	*6,1*	*5,4*

Tabela 17. Média de tópicos distintos e repetidos no top 10 dos resultados das pesquisas para a ordenação com o algoritmo BM25.

Combinação	*Média tópicos distintos@10*	*Média tópicos repetidos @10*
BM25 + Diversidade (SimHash)	*7,9*	*3,9*
BM25 + Aprendizagem + Diversidade (SimHash)	*9,1*	*3,1*
BM25 + Aprendizagem + Diversidade (MurmurHash)	*8,8*	*3,5*

Tabela 18. Resultados dos testes para o algoritmo de diversidade 1.

Combinação	*Média tópicos distintos @10*	*Média tópicos repetidos @10*
BM25 + Diversidade (SimHash)	*7,1*	*5,4*
BM25 + Aprendizagem + Diversidade (SimHash)	*7,9*	*4,4*
BM25 + Aprendizagem + Diversidade (MurmurHash)	*7,3*	*4,9*

Tabela 19. Resultados dos testes para o algoritmo de diversidade 2.

Para os dois algoritmos, utilizando a ordenação pelo algoritmo BM25 e seguida das reordenações por aprendizagem e por diversidade, obtemos os melhores resultados em termos de diversidade no top 10 dos resultados das pesquisas. Tendo o algoritmo 1 uma melhor média de tópicos distintos, sendo por isso o algoritmo escolhido para ser utilizado na reordenação das pesquisas do QSearch.

Mas comparando os resultados dos dois algoritmos de diversidade com o algoritmo BM25, verificamos que com o algoritmo de diversidade a média de tópicos distintos apresentados é superior em relação quando apenas é utilizada a ordenação pelo algoritmo BM25. Ora esta ordenação era a única ordenação utilizada no sistema QSearch, antes do início do desenvolvimento desta tese, e vendo a média de tópicos distintos quando é utilizada a ordenação pelo algoritmo BM25 e as reordenações por aprendizagem e por diversidade, constatamos que houve uma significativa melhoria na diversidade de informação apresentada aos utilizadores do QSearch.

Como era de esperar, face aos resultados obtidos no capítulo de similaridade documentos, quando comparámos os algoritmos *SimHash* e *MurmurHash*, quando o algoritmo de diversidade utiliza os *hashs* produzidos pelo algoritmo *SimHash*, é obtida uma maior média de tópicos distintos.

O algoritmo BM25 apresenta uma média de 6.1 tópicos distintos no top 10 dos resultados das pesquisas, sendo uma média mais baixa relativamente aos casos em que é utilizado um dos algoritmos de diversidade. Utilizando os algoritmos BM25, por

aprendizagem e diversidade 1 e 2 apresentam médias de tópicos distintos no top 10 dos resultados de 9.1 e 7.9, respetivamente.

A combinação do algoritmo BM25 com qualquer um dos dois algoritmos de diversidade apresenta médias de tópicos distintos inferiores em relação à combinação dos algoritmos BM25, por aprendizagem e diversidade, pelo que poderá ser interessante investigar como trabalho futuro, se existe alguma forma de melhorar a média de tópicos distintos utilizando a combinação do algoritmo BM25 e de diversidade, deixando de parte a reordenação por aprendizagem.

Face aos resultados obtidos, podemos concluir que o algoritmo desenvolvido aumentou a diversidade de informação apresentada nos resultados das pesquisas feitas ao sistema QSearch, sendo que este era um dos principais objetivos desta tese, pelo que foi conseguido.

5.5 Sumário

Neste capítulo apresentámos o conceito de diversidade de informação e a motivação para a utilização do mesmo no contexto desta tese. Para a construção do algoritmo de diversidade, baseámo-nos no conceito apresentado no capítulo 3, similaridade documentos.

Com a avaliação dos resultados obtidos para os algoritmos de diversidade, conseguimos concluir que com ambos os algoritmos desenvolvidos, a diversidade de informação no topo dos resultados das pesquisas aumentou face à diversidade de informação apresentada pelo algoritmo BM25, que era o método que anteriormente realizava a ordenação dos resultados das pesquisas.

6

Conclusões

O principal objetivo desta tese foi a exploração de técnicas para realizar a reordenação dos documentos retornados por uma pesquisa, através de uma análise mais cuidada da diversidade dos documentos.

Este trabalho foi realizado no contexto do sistema de pesquisas QSearch, já desenvolvido pela Quidgest. O sistema QSearch é um sistema de pesquisas, em que cada documento é apresentado com o seu título, categorias a que pertence, as anotações feitas pelo utilizador e os documentos relacionados com o próprio, nomeadamente, os seus duplicados e similares. A figura 20 ilustra a interface atual do QSearch.

Nesta tese foram exploradas duas áreas: a **reordenação por aprendizagem** e a **reordenação através de uma análise da diversidade de informação no topo dos resultados das pesquisas**. Para a reordenação por aprendizagem foram estudados alguns algoritmos, de forma a escolher o melhor para este problema específico. Para a reordenação através de uma análise da diversidade de informação no topo dos resultados das pesquisas foi utilizado o conceito de similaridade documentos, apresentado no capítulo 3, de forma a identificar e retirar documentos similares do topo dos resultados das pesquisas.

Com o conceito de similaridade de documentos, utilizando algoritmos de *locality sensitive hashing*, introduzimos no sistema QSearch um novo mecanismo para eliminação de documentos similares/duplicados no topo dos resultados das pesquisas, de forma a aumentarmos a diversidade de informação.

O trabalho desenvolvido melhorou as capacidades de reordenação do sistema QSearch, mais concretamente em termos da reordenação dos resultados das pesquisas,

na medida em que o algoritmo de diversidade aumentou a diversidade de informação no topo dos resultados das pesquisas, utilizando também algoritmos de reordenação por aprendizagem, como foi comprovado nos resultados do capítulo 5.

Como trabalho futuro poderiam ser exploradas abordagens alternativas para identificar documentos similares, desenvolvendo novos algoritmos de diversidade através dessas abordagens. No contexto da reordenação por aprendizagem, poder-se-ia analisar o impacto de novas características de aprendizagem nos dados da Quidgest.

Referências

[1] "Apache Tika" [Online]. Disponível: http://tika.apache.org/.

[2] "Apache Solr" [Online]. Disponível: http://lucene.apache.org/solr/.

[3] "RankLib" [Online]. Disponível: http://people.cs.umass.edu/~vdang/ranklib.html.

[4] "Apache Lucene Core" [Online]. Disponível: http://lucene.apache.org/core/.

[5] C.J.C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton and G. Hullender. Learning to rank using gradient descent. In Proc. of ICML, 89-96, 2005.

[6] Y. Freund, R. Iyer, R. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. The Journal of Machine Learning Research, 4, 933-969, 2003.

[7] J. Xu and H. Li. AdaRank: a boosting algorithm for information retrieval. In Proc. of SIGIR, 391-398, 2007.

[8] "Secure Hash Algorithm". [Online]. Disponível: http://en.wikipedia.org/wiki/Secure_Hash_Algorithm.

[9] "Distância de Hamming" [Online]. Disponível: http://pt.wikipedia.org/wiki/Distância_de_Hamming.

[10] Z. Cao, T. Qin, T.Y. Liu, M.F. Tsai, and H. Li. Learning to Rank: From Pairwise Approach to Listwise Approach. ICML 2007.

[11] "Apache PDFBox - Java PDF Library" [Online]. Disponível: http://pdfbox.apache.org/.

[12] "Apache POI - the Java API for Microsoft Documents" [Online]. Available: http://poi.apache.org/.

[13] "Neko HTML" [Online]. Disponível: http://nekohtml.sourceforge.net/.

[14] "The size of the World Wide Web (The Internet)" [Online]. Disponível: http://www.worldwidewebsize.com/.

[15] Liu, T.-Y. "Learning to Rank for Information Retrieval. Foundations and Trends in Information Retrieval". 3, 225-331, 2009.

[16] F. Xia and J. Wang, "Listwise Approach to Learning to Rank - Theory and Algorithm", 2008.

[17] "OHSUMED Test Collection" [Online]. Disponível: http://ir.ohsu.edu/ohsumed/ohsumed.html.

[18] D. Nemirovsky, "Web graph and PageRank algorithm", JASS Conference, 2005.

[19] Robertson, S.E. and Walker, S. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. 232–241, 1994.

[20] Salton, G. and Buckley, C. Improving retrieval performance by relevance feedback. Journal of the American society for Information science. 41, 288 – 297, 1990.

[21] Manning, C.D. et al. Introduction to Information Retrieval. Cambridge University Press. 2008.

[22] W. Hersh, C. Buckley, T. J. Leone, and D. Hickam. Ohsumed: an interactive retrieval evaluation and new large test collection for research. In SIGIR '94, 192–201, 1994.

[23] S. Büttcher, C. Clarke, and G. Cormack, Information Retrieval Implementing and Evaluating Search Engines. 455-460, 2010.

[24] "Information retrieval" [Online]. Disponível: http://en.wikipedia.org/wiki/Information_retrieval.

[25] Qin, T. et al. 2010. LETOR: A benchmark collection for research on learning to rank for information retrieval. Information Retrieval. 13, 346–374, 2010.

[26] "MD5" [Online]. Disponível: http://en.wikipedia.org/wiki/MD5.

[27] "MurmurHash" [Online]. Disponível: http://en.wikipedia.org/wiki/MurmurHash.

[28] J. Kleinberg, Authoritative sources in a hyperlinked environment, Journal of the ACM (JACM). 604–632, 1999.

[29] "Distância Levenshtein" [Online]. Disponível: http://pt.wikipedia.org/wiki/Distância_Levenshtein.